Definition of a Distributed Machine Health Monitoring System: Application to the Design of a Hydraulic Marine Steer-by-Wire System

by

Nicholas Eoghan Cullingham

B.Sc., University of Calgary, 2001

A thesis submitted in partial fulfilment of the requirements for the degree of

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate Studies
(Mechanical Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

January 2006

Abstract

The objective of this thesis is to define a general tool set for monitoring the health of the physical part of a system that is operated by an embedded computer system. Embedded computer systems are commonly found in industrial, military, transportation and consumer products. Health monitoring is usually required to track the health of actuators, sensors, communication, and computational resources.

The novel tool set presented in this thesis provides three measures of component health, all of which indicate errors based on the performance of a system state relative to a base model. A set of software classes is defined, which interacts with an object-oriented model of the physical system that supports distributed processing, fault tolerance and redundancy management. The model based health monitor also integrates into the redundancy management. When a redundant physical sensor fails, the health monitor provides analytical redundancy for that system state, by predicting it and generating adaptive thresholds on the accuracy of the analytical sensor.

The health monitoring system has been implemented on an experimental apparatus built to approximate the functionality of a hydraulic, steer-by-wire system for marine applications.

Contents

List of Tables

List of Figures

List of Symbols and Acronyms

Δ	discrete time interval
h_d	system dynamics health indicator
h_i	instantaneous error health indicator
h_a	average state change health indicator
$P_{k\|k}$	the error covariance matrix at a time, *k*, given the state *k*
k	system gain. also a single discrete time interval (for the EKF)
T_D	torque measured at the drive
T_H	torque measured at the helm
θ_H	rotational position of the helm
θ_D	linear position of the drive
u	general input function of time, t
V_D	command signal sent to the drive actuator
V_H	command signal sent to the tactile feedback device
ω_n	undamped natural frequency
ω_{lb}	lower boundary of ω_n
ω_{ub}	upper boundary of ω_n
$\hat{x}_{k\|k}$	the estimate of the system output state at a time, *k*, given the state *k*
y	current system output state with respect to time, t
$\hat{y}$	estimate of the system output state at a time, t
$\hat{y}_{ub}$	predicted upper bounding threshold at a time, t
$\hat{y}_{lb}$	predicted lower bounding threshold at a time, t
ζ	damping ratio
ζ_{lb}	lower boundary of ζ
ζ_{ub}	upper boundary of ζ
ARI	Average Response Indicator
DED	Distributed Error Detection
DRI	Dynamics Response Indicator

ECU	Electronic Control Unit
EKF	Extended Kalman Filter
FSU	Fail-Silent Unit
FTU	Fault-Tolerant Unit
HARP	Helm Axis Rotary Position
HARV	Helm Axis Rotary Velocity
IRI	Instantaneous Response Indicator
LED	Local Error Detection
PWM	Pulse Width Modulated

Acknowledgements

I would like to acknowledge and thank my research supervisor, Dr. Ian Yellowley, for his guidance throughout my program. I have found my experience both enriching and enjoyable, because of his academic expertise and financial support during my Master's program.

I would also like to thank my lab partners, Mathieu Bouvier and Kevin Oldknow, for their work, advice, and ongoing support.

As well, I would like to thank my parents, Valerie and Owen Cullingham, for their moral support, time, and generosity. Finally, I would like to thank my wife, Andrea, for her unwavering patience and encouragement, without which I could not have completed this work.

- Nicholas Cullingham

Chapter 1: Introduction

1.1 Embedded Computer Systems

Embedded computers and sensing systems already play a crucial role in the control of a wide variety of applications. Consumers demand the flexibility and options that integrated computing systems can provide, so one may expect that the level of use will continue to increase.

The current generation of embedded computers provide control functions in applications that range from the critical and hard real time requirements of aircraft to consumer products such as washing machines. Typically, these embedded computers are low-power devices with computational capabilities that are matched to the minimum functional requirements of the devices that they control. A number of benefits can be gained simply from the integration of embedded computers in devices; this is particularly the case when excess computational resources are used to enhance the product.

One of the largest potential benefits is performance improvement. Computing systems allow the development of more powerful, flexible, and efficient control strategies, which allow for the closed loop response to be tuned. Consumer devices can be tuned for each individual user, or to a designer's specification. For example, consider the design of a traditional automotive steering system which consists of a mechanical linkage and hydraulic power steering: At low speeds, turning the steering wheel requires a large effort and greater range of motion than when driving at highway speeds, where a lesser effort and small range of motion affect a significant steering effect. In a steer-by-wire system, the vehicle's speed can easily be included within the controller, allowing for speed dependant, variable gain steering.

The second major area of benefit is in physical design flexibility. Consider the automotive steering column. Replacing the steering system with an embedded computing system allows for more options in the design of the engine compartment, and it also makes the vehicle structural and engine design identical for left and right hand drive; only the user interface needs to change. This flexibility provides a large saving in manufacturing cost, more choice in the design of the steering input system, and allows designers the freedom to select configurations that take into account greater user safety in event of a collision.

Finally, integrated computer systems allow for the inclusion of new features. Sensor data can be collected and stored, tasks can be automated, and system health and performance can be monitored. The system monitor can record current performance and check for errors, which in turn allows for diagnostic capabilities. Monitoring functions allow the provision of fault-tolerance, and also condition-based predictive maintenance, which can increase up-time and extend product life.

1.2 ***The Reason for Safety Critical Design***

A safety critical function is a function that cannot fail in a safe manner; any unhandled error[1] that occurs will lead to a situation which causes elevated risk to the

1 Any unhandled error is considered a failure. See section 2.3.

system, its users, or environs. For example, a bass sport boat can achieve speeds over 160 km/hour (100 mph). A boat could be passing through shipping lanes at such speeds when an error occurs within a component of the steering system. There is no fail-safe position for the steering system, because no default position exists that the system can home to. Because the system requires continuous, uninterrupted human intervention to mitigate risk, the system must provide the human user with the correct information at the correct time, and then respond in the correct way for the safety of the boat driver. One of the risks of using embedded computing systems is that when an electronic component fails, it tends to be binary: either it works, or it does not.

1.2.1 Handling Failure

Guarding against the failure of high-level service assurances has been the focus of research and applications over the past 20 years. All of these efforts have focused on reacting to failures that could occur within the system, and maintaining high-level services despite some component failures. The measure used to describe success is the reduction in probability of cascading or high-level service failure. All of these efforts have focused on designing a safety framework to provide reliable, continuous function of high-level services. Each system is designed to detect component failure, and when it does, switch to a back-up system.

1.2.2 Monitoring Performance

Monitoring the behaviour of actuator components allows one to detect normal and abnormal behaviour. Establishing a metric for the health of a component based on expected behaviour allows a system to detect errors and identify operation that may indicate component wear or impending failure. Normally, monitoring requires the addition of new sensors to measure signals which directly indicate system performance. The normal approach to condition monitoring incorporates indicators of performance that provide clear and unambiguous information about specific faults. These indicators are usually derived from modelling or experimentation. Condition monitoring is a necessary part of the failure handling framework, and provides some of the tools required to use and manage redundant actuators.

1.2.3 Challenges

Despite the potential benefits outlined in section 1.1, the design of an integrated embedded system must consider the system's ability to cope with faults. The greatest challenge in design is to guarantee that the system will perform safely even if one or more components fail. The scope of safe performance is not limited to the user, but extends also to the safety of others and their environs; it is further required to limit liability and gain public acceptance of embedded computing systems. Existing technological solutions to the problem are dependent on high levels of redundancy and have high costs associated with them. This type of solution is not appropriate for the development of products in cost sensitive markets such as consumer devices, automobiles, and pleasure boats, nor is it ideal for industrial or military applications.

1.3 Thesis Objective

The objective of this thesis is to examine the possibility of providing generic tools to aid in the indication of the health of system components. The methods defined should make use of multiple layers of redundancy, and be aware of redundant sensors and actuators. Furthermore, they are required to interface with a highly object oriented virtual environment. The tools then will provide both lower level methods that can be customized to each use and each environment, as well as the higher level predefined health monitoring methods which apply more generally.

The secondary goal of the system is to add analytical sensor health detection to the fault-tolerant framework that will protect the sensor fault-tolerance mechanism against continuous degradation following a sensor failure. In a fault-tolerant sensing system, a single value is determined from a set of sensors. When one sensor fails, the arbitration mechanism is weakened: this is continuous degradation. If sufficiently weakened to only include two physical sensors, the arbitration mechanism cannot detect Byzantine errors between the two sensors. An appropriate software indicator can help to arbitrate between two sensors, but only if the model used can be verified to accurately represent the current parameters of the process being modelled. The proposed health monitoring system has the benefit of verifying the accuracy of the system model before using it to verify sensor integrity.

1.4 Thesis Outline

This thesis is organized into five chapters. Chapter 2 is a literature review, and discusses the existing technologies that are used in health aware systems. The review concentrates upon and discusses the application to marine steering systems, fault-tolerance, condition monitoring, modelling and diagnosis.

Chapter 3 gives the context of the health monitoring system, and describes the approach taken. First, it describes the existing fault-tolerant framework, and then it describes the approach taken to create the health indicators, what they represent, and why each of them is included. The development and mathematical analysis of the indicators are then presented. Finally, it describes the object-oriented systems architecture that contains the health indicators.

Chapter 4 describes the specific implementation of a health monitoring system to the marine hydraulic steer-by-wire test system that was used to demonstrate the concepts developed within this thesis. This is followed by the results obtained from the experiments that were performed on the test system. An analysis of the data shows how the health indicators provide useful information about the health of the system, and that the analytically redundant sensor adequately predicts future sensor state values. It closes by discussing the achievements and limitations of the indicators as presented.

Chapter 5 presents the overall conclusions of the work and makes recommendations for further work.

Chapter 2: Health Aware Systems

2.1 *Introduction*

This chapter provides a literature review of topics that relate to health aware systems. It describes key concepts that relate to the fault-tolerant systems that provide the framework upon which this health monitor is developed. It shows the limitations of current research, and how work in the health monitoring field can enable other achievements.

The design of health indicators is usually derived based on modelling or experimentation, which makes them application dependent. The concepts developed within the thesis are demonstrated on a marine hydraulic steer-by-wire system, so this chapter also reviews current marine steering systems, and faults common to the steering demonstrator.

2.2 *Marine Steering Systems*

A number of manual steering systems exist, from mechanical linkages to hydraulic circuits. Each of these systems may provide some mechanical advantage to the user, or the user's input may be augmented through power steering systems. Recent advances have integrated closed loop navigation with the steering system to provide autopilot features. Each system has its advantages, and is prone to particular errors. Steer-by-wire systems provide a new set of challenges and potential faults to be considered during their design.

2.2.1 Steering Systems

Manual

The most straightforward steering systems are of the manual variety; the simplest of these is likely the cable steering system. In this type of system, the operator turns the helm, pulling the cable to impose a motion of the rudder. This type of system is well suited to low-cost, low-power pleasure boats and provides a durable, reliable system which requires little maintenance. The number of turns of the helm, lock to lock, determines the effort and responsiveness of the rudder. A high number of helm turns occurs with a high gear ratio; this leads to less effort (torque) needed at the helm, but yields a small response at the rudder. Likewise, a lower gear ratio leads to a greater effort needed at the helm, but provides a faster response at the helm.

Hydraulic Steering

Another very common steering mechanism uses a hydraulic connection from helm to rudder. In this system, a manual pump is part of the helm and is connected via hoses to a steering cylinder that provides the movement at the rudder. Hydraulic steering systems are usually found in larger, more powerful pleasure boats. The use of hydraulic fluid tends to lead to more frequent maintenance than mechanical systems, but they usually provide a smoother, more stable steering mechanism than the mechanical linkage. An example of this type of steering system is shown in Figure 2.1.

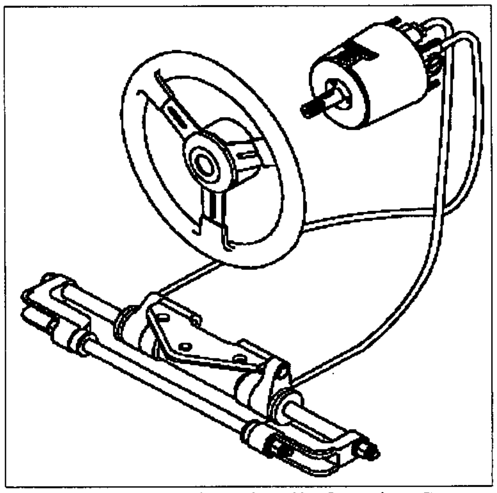
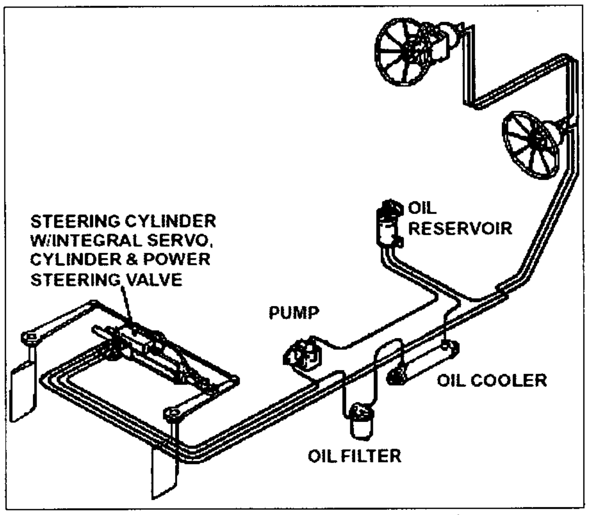


Figure 2.1: Manual Hydraulic Steering System
© Teleflex used with permission

Power Steering

The hydraulic steering system can also be augmented by including power assistance. In a power steering system, a hydraulic pump, driven by either the engine or an electric motor, provides an assistive force on the steering cylinder. The primary steering system acts as described previously, but now the steering cylinder is fitted with a servo cylinder and a power steering valve. The valve opens when the manual cylinder moves, powering the secondary cylinder to provide the desired assist. Should the power assist circuit fail, the system automatically falls back to a traditional hydraulic steering mechanism.

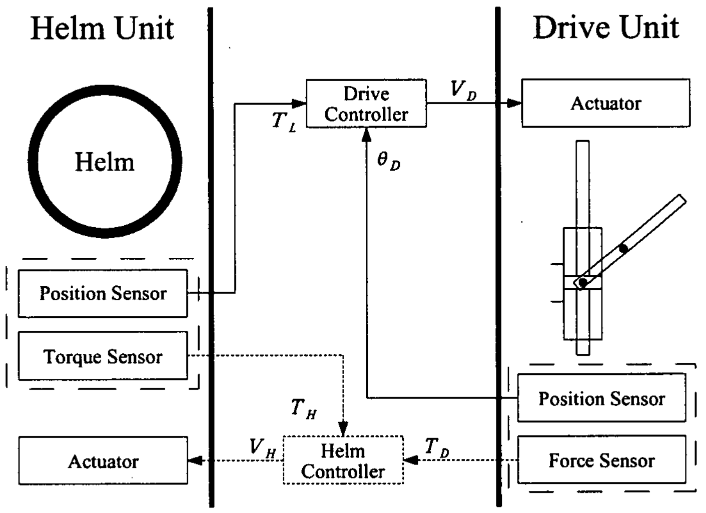
Alternatively, a more advanced power steering system could be used, which uses position or pressure sensors at the helm, and connects to a digital control system to regulate a power assist valve or the automatic pump in the secondary hydraulic circuit. In this type of system, the user could tune the power steering control system to vary the amount of assist provided. In Figure 2.2 an example of the hydraulic power assist steering system is shown.


STEERING CYLINDER W/INTEGRAL SERVO, CYLINDER & POWER STEERING VALVE
OIL RESERVOIR
PUMP
OIL COOLER
OIL FILTER
Figure 2.2: Hydraulic Power Assist Steering System
© Teleflex used with permission

Steer-By-Wire

In a steer-by-wire system, no mechanical or conventional direct hydraulic connection exists between the helm and the rudder. Instead, a digital controller measures commands from the user and electronically controls an actuator at the rudder. By definition, a steer-by-wire system is a steering system in which the hydraulic or mechanical connection between the helm and rudder is replaced by an electronic control.

Unlike other forms of steering systems, steer-by-wire can be tuned not only to each user's preferences, but also to changing boating conditions. The digital control algorithm can be flexible and based on information that is not normally incorporated into a typical steering system. It may also be desired to implement tactile feedback, where the forces acting on the rudder are mirrored at the helm. A diagram of how such a feedback system might look is shown in Figure 2.3.



θ_D	linear position of the drive
θ_H	rotational position of the helm
V_D	command signal sent to the drive actuator
T_D	torque measured at the drive
T_H	torque measured at the helm
V_H	command signal sent to the tactile feedback device

Figure 2.3: Steer-By-Wire Schematic

The primary steering function is provided by the drive controller, and the feedback at the helm is provided by the helm controller. The drive controller implements closed loop position control, and the helm controller may also provide an open loop or a closed loop torque control system. Position, torque and force sensors are used for these control systems.

Usually, the embedded computer that manages the sensor acquisition, control, and actuation is chosen to minimize cost subject to it having the minimum of processing capability required to implement the steering control loop. One could expect a dedicated micro-processor based computer to operate between 10 and 100 MHz, in typical cases.

2.2.2 Hydraulic Actuation

Marine based steer-by-wire systems often achieve movement of the rudder through a digitally controlled hydraulic circuit. There are a variety of ways of implementing a hydraulic circuit, one possible implementation is shown in Figure 2.4. This figure shows a straightforward hydraulic positioning circuit; variations on this circuit may include a variable flow pump, or a pump driven by the boat engine; the accumulator could be removed, hoses could be replaced by tubing, or valves could be combined. This circuit is digitally controlled at the 4-way, 3-position directional valve, by simple state control (bang-bang), and at the motor, also by on-off state control (bang-bang). In a more sophisticated system, the pump is connected to a DC motor which could then be controlled to provide variable velocity. Another enhancement to this system would replace the directional valve with a proportional valve.

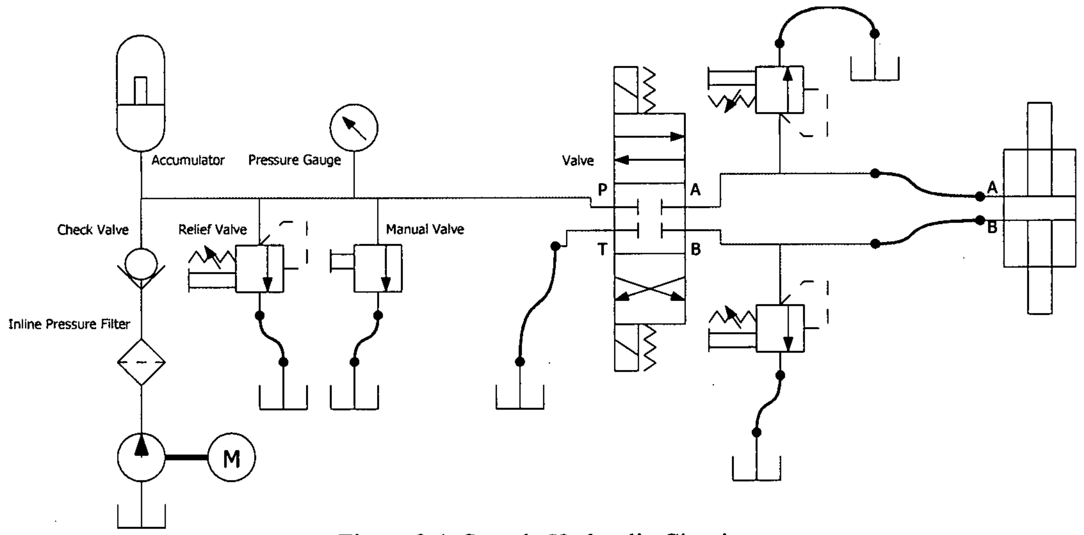


Figure 2.4: Sample Hydraulic Circuit

The steering cylinder itself is a double acting cylinder, providing symmetric performance in each direction. Relief valves guard against exceeding a set pressure maximum within the system.

2.2.3 Common Errors

The behaviour and performance of hydraulic systems is dominated by their dependence on pressure and flow rate [1]. Fluid pressure measurement is readily available through reliable pressure transducers (a change in pressure drop across a component could be an indication of wear, performance degradation, or impending failure). Likewise, a change in flow rate into and out of components might also indicate a fault, however measuring flow rate through components is unfortunately, very difficult. A number of fault conditions can affect pressure and flow rate.

Examining the hydraulic circuit shown in Figure 2.4 reveals a number of components where errors could occur. When considering errors in a control system like the steering system, it may be beneficial to detect errors at varying degrees of granularity. Some localized component faults present little risk to the system over the short term, while other faults have the potential to severely hamper performance of the entire system.

A number of fault conditions may affect pressure and flow rate; the basic areas of concern are the following.

- **Fluid**: A number of potential fluid faults exist. Contamination can be a fault or a symptom of another fault. Particulates, solutes, and air may become dissolved within the hydraulic fluid. Symptoms of fluid contamination by particulates or solutes may include elevated temperature, and a change in flow rate; the contamination can clog filters, cause or be the result of component wear, and change fluid viscosity. In addition, contamination due to a compressed gas will cause pressure loss. The risks associated with fluid contamination include accelerated wear, decreased performance, and system overheating.

- **Motor**: Faults at the motor directly affect the pump. Symptoms of low current or voltage at the motor are a reduction the flow rate or a decrease in pressure provided by the pump. The critical risk associated with motor power loss is the loss of system functionality.

- **Pump**: The pump provides the pressure and flow rate for the system. With a number of moving components, and multiple interfaces to the fluid system, the pump is vulnerable to a number of faults. Directly measurable symptoms include leakage and fluid temperature, while risks associated with pump faults include component wear, friction, over-pressure and under-pressure performance.

- **Connections**: The main fault that connections are subject to is leakage. One symptom of leakage is increased flow rate through the component, or a decreased flow rate downstream. The main risks associated with poor connections are contamination, pressure and fluid loss, and if not monitored, draining of the reservoir.

- **Accumulator**: Over-charged or under-charged accumulators can both affect system dynamics. An over-charged accumulator will have very little impact on the system, and an accumulator in the system is usually designed to supplement flow rate or stabilize pressure about a set point. Likewise, a drop in charge pressure will prevent the accumulator from stabilizing pressure and flow rate. Often, accumulators are included to dampen high frequency fluid shock. So key symptoms of accumulator error would be higher frequency fluid shock, or higher power drain at the pump/motor. Risks caused by accumulator faults include high system shock, leading to rapid wear, and increased peak flow demand at the pump.

- **Valve**: Faults associated with the valve include spool wear and friction. Symptoms of spool faults could include slow system response, decreased flow rate, and incorrect flow direction.

2.3 *Fault-Tolerant Systems*

As discussed in Chapter 1, steering systems demand that a safety-critical design approach is taken. A number of research projects have investigated the implementation of fault-tolerant systems. Hiller [2], provides a summary of terms used in fault-tolerance.

- A **fault** exists when there exists a state of operation for a system that leads it to non-conformance of its specifications. Faults are classified by their duration, as either transient or permanent, and their realization, as dormant or active.

- An **error** is the manifestation of an active fault; it is an occurrence of the system entering a state of non-conformance to its specification. Undetected errors are called latent [3]. The key difference between a fault and an error is that where errors are the results of faults, only an error is measurable.

- A **failure** is the result of an unresolved error. Note that a failure of a low-level system might be considered a fault by a higher level system, which could in turn trigger a different error.

Fault-tolerant design also considers the level of faults. In a fault-tolerant system, low-level errors may occur at a small locality, and may be acceptable provided the system has mechanisms to avoid high-level system failures. The first important step in responding to an error is error detection. Koopman [4] has performed extensive work in reconfigurable systems and graceful degradation[2], and provides a description of system response and recovery in a number of reconfigurable environments, which he terms the self-healing problem space. A self-healing system is the extension of fault-tolerance to a system that can recover functionality, lost after an error has occurred.

Errors can occur in the value domain, and also in the time domain. An error in the value domain occurs when an actual value (measured, estimated, or communicated), differs from the expected (or allowed) value. An error occurs in the time domain when a value is reported at the wrong time, or not at all at the expected time. In a hard real-time application, the system must always produce the correct value at the correct time. Errors can occur in both domains simultaneously, or either one.

2.3.1 Error Detection

Error detection can take on different forms, but these are generally classified as either data replication or executable assertions. Data can be replicated through analytical redundancy, double execution, and hardware replication. Executable assertions include limit checking, certification, signature checking, self-tests, and watchdog timers [5]. Furthermore, detection can be performed within a component or via a network by peer-to-peer checking or supervisory elements. Distributed environments require a mechanism to synchronize their timings, and when error detection is performed in a distributed environment, errors may be detected in the time domain: each computer can check the communication timing of the other computers.

Redundancy is a key component in error detection. To detect an error in any component, there must be a redundant component or task to form a basis for comparison;

2 Graceful Degradation: see section 2.3.3

to identify the faulty component, more information about the components or further redundancy is required.

When using analytical redundancy, the acquired sensor data is assumed to have provided accurate data previously, and is compared at the current time step to a model of the system. The difference between the estimated state and the actual state is referred to as a residual [6], or a bias.

One form of executable assertion uses *a priori* modelled actuator, plant, and sensor errors [7]. Each error model can be included as part of a model of the larger process, acting as a filter to detect errors. Each error filter model is populated with the known system states and inputs, and becomes non-zero when that error is detected.

When using physical redundancy, a single state is measured by more than one sensing element, and these values are compared via voting techniques. A wide variety of voting techniques exist, and are classified by their input and output mechanisms [8]. Input mechanisms can be exact or inexact, which concerns whether the input data is considered inflexible (exact) or flexible within a region (inexact). Outputs are classified as consensus or compromise, referring to whether the output is selected from one of the inputs (consensus), or is calculated by a median or mean (compromise). Output determination mechanisms also can be classified as threshold or plurality. Threshold voting checks whether the selected value has achieved a predefined threshold quantity of votes, like unanimity, Byzantine (quantity > 2/3 of the votes), majority, m-out-of-n, etc. Plurality voting selects the value with the most votes [9], or if no value has more votes, then the output value can be non-unique.

Within a steer-by-wire application that uses Electronic Control Units (ECUs), the controllers must also be checked for errors. Checking a processor for errors is achieved through redundant controllers, forming a network. This network allows the creation of a distributed computing environment, where each ECU shares local information with its peers, and each ECU can check its peers for errors.

2.3.2 Granularity

The **granularity** of a system is the extent to which it is composed of separate components, and of each component's functional "size" relative to the entire system. Therefore, a system with finer granularity has more components, and each one accounts for a smaller increment of the whole.

It is important to match the granularities for fault detection and fault isolation in the design of the system. Providing a very fine granularity increases system complexity (because the number of components increases), but allows each component to be represented in a straightforward manner, with simple models.

One could also design the system with a coarse granularity, where the developer would use a set of state equations to describe each component. Providing a very coarse granularity decreases the complexity of the overall system description, but the models that describe each component become more complex. It also increases the complexity of detection and diagnosis of faults relative to what would be required for a fine granularity.

The fault isolation mechanism cannot have a finer granularity than the fault-detection mechanism. However, one could design the system to achieve a fine granularity in the health monitoring and fault detection mechanism, where the states and parameters of each component are monitored. Supposing that the output from the health indicator for a single component is a scalar value, then for a group of components, the health indicators could be grouped into a composite health vector. The fault-isolation mechanism would be aware of the health vector representing the constituent components and could take appropriate action to isolate an error.

Using a fine granularity requires the same or more design knowledge than the coarse granularity system, depending on the types of errors that are to be detected, and the granularities of the fault-isolation and the error-diagnosis mechanisms. It makes each component health indicator more straightforward and less computationally intensive than the coarse granularity case. It also provides the benefit of being able to detect and localize faults more precisely. Finally, each component health indicator is an independent task, which enables the use of distributed processing.

The key disadvantage that one must balance against is that in order to achieve a fine granularity of system faults, one requires an equally large set of observed system states. Increasing the number of observed states adds physical sensors to the system, which escalates system cost.

2.3.3 Graceful Degradation

To cope with failures, a system may reduce performance capabilities, cancel less important tasks, or switch to different control algorithms. This method of tolerating errors is called **Graceful Degradation**. Arguably, any fault tolerant system that masks or isolates failing components undergoes degradation, as it loses the redundancy supplied by those components. The key identifier of graceful degradation is that when a component or function fails, it is replaced by a non-identical component or function of reduced capability.

The Robust Self-configuring Embedded Systems (RoSES) project [10], uses data and control flow graphs to reconfigure its software. The described system can handle multiple failures and attempts to optimize system utility.

An alternate approach, described by Bouvier [11], uses diverse and exact redundancy to handle failures, which makes low level failures transparent to high-level methods. Also, the approach uses a table of known methods, called an execution table, to select methods at run time that provide redundant functionality.

2.3.4 Error Response

Error response refers to the manner in which a system reacts once an error is detected. Common responses include masking, rollback to system checkpoint, and re-trying the operation, possibly using substitute resources. More powerful responses include architectural reconfiguration, alternate execution paths, degradation, and requesting assistance. Already mentioned are the RoSES and Amaranth projects. Remedies involving simple detection include change of operation, reconfiguration and stopping the process. With a diagnostic capability, the system can then determine if the failure is conditionally tolerable, it can reconfigure the process parts, or request specific maintenance.

The body of work in fault-tolerant control is extensive. One technique for achieving fault-tolerant control is to use state-estimation techniques to find a bias between expected and actual states, and use this estimated state in the control law [12] [13].

The X-By-Wire project [14], defined an entire fault-tolerant architecture. It used an approach of exact redundancy and fail-silence throughout the project. The project involved a number of members from the automobile industry and from academia. They designed a system in which any communicating component must be a fail-silent unit (FSU). In a FSU, the component must be able to self check and output either the correct value or nothing at all. Furthermore, each atomic subsystem is actually a fault-tolerant unit (FTU), which is composed of two exactly redundant FSUs. When an error occurs, the units ensure that only a correct output is realized. One problem with this system is that to support error detection of any one ECU, two ECUs are needed, making the system cost inefficient. One example of actuation was realized by having three redundant DC motors connected to a specialized gear box. Each actuator would provide one third of the required torque in normal operation, and half of the required torque in an error state. The report concluded that it was more cost effective to have three motors capable of outputting 50% of the required torque than two that could output 100% of required torque.

Bouvier [11] developed a distributed fault-tolerant architecture and demonstrated it in a steer-by-wire application. This system uses triple-modular redundancy and inexact voters to detect data errors. Each ECU could perform local error detection on the data it acquired, and also a distributed error detection with its peers to detect sensor, ECU, and communication errors. The system response strategy uses a dynamic reconfiguration approach, in which the system can reconfigure resources at run-time. This architecture uses an abstraction of system hardware and resources in combination with an execution task flow chart to select tasks. The fail-silence property described by the X-By-Wire project is also implemented on each ECU. Each ECU is connected to a second set of channels called state-lines, which operates as a voting mechanism, where two peers must validate the local ECU as being functional to allow it to broadcast data over the communications bus. This thesis uses and enhances the fault-tolerant architecture developed by Bouvier.

Das [15], evaluates the effectiveness of various fault management architectures' response to task and CPU errors in distributed computing environments, and validates a distributed peer architecture as having a high number of reachable configurations to use when executing reconfiguration and tasks.

2.3.5 Recovery

The **recovery** operation is an enhancement to fault-tolerance, found only in self-healing systems. It involves the integration of new resources to the systems that are executing tasks to return it to an error-free state.

A component might be hot swapped; the best example of a fault-tolerant system that supports hot swapping is the RAID array of hard disk devices, in which a failed disk can be replaced without rebooting the system. Alternative approaches involve reconfiguring software to match newly installed components, or self discovery of new components at soft-reboot and hard-reboot.

2.4 *Health Awareness*

The majority of work in fault-tolerant systems discussed so far relates to the detection and handling of errors, where errors are classified as existing in a binary state: either the error exists (the system is in a state of non-conformance with specification), or the error does not exist. A weakness with the binary error approach of these fault-tolerant systems is the lack of error predictability. The ability to predict impending failure allows the operator or system to take pre-emptive action to prevent failure by altering system functionality or scheduling maintenance. Indicators of system health give insight into the likelihood of failure.

To identify whether a system's health has degraded to the point where failure is impending, a number of indicators can be monitored and compared to alarm limits. Alarm limits can be adjusted according to factors including experience, supplier recommendations, previous failure data and standards.

Health indication seeks to represent the system's health state by selecting indicators which describe system performance or correlate to known failure modes. There are a number of approaches that have been used to develop health indicators, which can be classified as either Condition Monitoring, Parameter Identification, or State Estimation. These three approaches are discussed throughout the remainder of this section.

2.4.1 Condition Monitoring

In **Condition Monitoring**, system states are measured that are not necessarily used in the control of the system. While the states measured may impact system performance, their effect is usually not included within the control model as modelling parameters or state variables. A variety of internal system parameters may indicate errors; temperature can reveal lubricity problems, misalignment, or overload; noise may indicate cavitation, valve, gear, or bearing wear. Internal parameters are discussed in, Fitch [16]. The use of these internal states to predict failure is discussed in, Goode [17], who proposes a prediction model theory based on statistical process control, and a failure model using a stable zone and failure zone to predict time to failure.

Noise and vibration are states that are also directly measurable, and can be good indicators of component wear, but are not usually accounted for within system models. In

Yunbo [18], the amount of vibration energy within a hydraulic system was found to indicate system wear; the analysis used Fourier transform techniques to analyze the vibration energy of the system. A newer technique which is gaining attention for condition monitoring is use of the wavelet transform. The wavelet transform can use a short window (high time resolution) in the time domain to represent high frequency content, and use a long window in the time domain to represent low frequency content, allowing for greater flexibility of resolution. The wavelet transform could be viewed as a signal decomposition into a set of basis functions [19]. The prototype of the basis function is scaled in the time domain. The use of a prototype basis function also means that the transform is not a mapping of the time-frequency plane, but a time-scale plane where the scale is dependent on the basis function.

The wavelet transform is used by Wang [20], to detect error conditions in a gearbox. He found that damage to a gear tooth causes a change in the vibration signal associated with the period of that tooth's engagement. Results from this work were presented only graphically, and automated interpretation was not discussed. The wavelet transform outputs a large amount of data, and to date, there is no mature method to interpret the data for content that specifically indicates health. However, Luo [21], has shown that the peak values from specific frequencies might be used as indicators of health.

Yang [22], discusses using bispectral analysis, and compares it to the wavelet transform. The bispectral analysis is a probability analysis showing the coupling between three frequencies, k, l, and m ($=k+l$). It also retains magnitude and phase information. Key data points from the bispectral analysis and wavelet transform were identified using a singular value decomposition, scaled, and passed to an artificial neural-network to classify faults.

2.4.2 Parameter Identification

Parameter identification techniques seek to determine and monitor the model parameters for the system over time; a common way to do this is to use observers. Usually, a system model of a given order is defined, and then a least squares regression is performed to determine the parameters for the given relationship between system input states and measured system output states. The system parameters can then have limit values applied to them that describe acceptable operation.

Isermann [23], describes the parameter estimation method for linear systems and how significant parameter changes can be detected by reference to the normal values using statistical methods like the Two-Probe T-Test.

One weakness in this method is that the parameters of a given order model may not represent system characteristics in a predictable way. Typically, these systems would then need to be developed for each application experimentally, and the limits must also be set experimentally. This reduces the potential design benefits of using a system model. However, in Yu [24], the lumped parameters obstacle was approached by investigating the change in each parameter grouping and comparing it to the expected composition of the lumped parameter set. Examining which parameters were present in each lump and comparing which lumps changed, allowed that work to infer which parameters were changing, and by how much they changed.

As part of the investigation of approaches to monitoring system performance, this approach was tested on the hydraulic steering system built by the author. The tested observer used second order, third order, and fourth order models, with Plackett's algorithm for recursive least-squares system identification. It was found that when a square wave was input to the system, the parameters tended to converge on a single model. However, further testing found the model parameters to fluctuate significantly depending on the form of the input, and in some cases did not converge at all.

2.4.3 State Estimation

The process of **state estimation** involves using a known system model, and applying a set of inputs to known system states to predict the output state at a given interval. The health indicators that are built from state estimation are generally composed of the residual error between the predicted state and the measured state.

Extended Kalman Filter

One technique gaining widespread use is the **Extended Kalman Filter** (EKF). In general, the EKF is a recursive filter that estimates a complete system state vector from a noisy or incomplete measured state vector. It relies on the developer to create a robust model of the system that describes system behaviour for all states, and then linearizes that model about each current operating point; the linearization allows the application of other linear algebra techniques. The state of the filter is represented by two variables:

$\hat{x}_{k|k}$, the estimate of the system state at a time, k, given the state k

$P_{k|k}$, the error covariance matrix at a time, k, given the state k

The filter can be thought of to operate in two distinct phases. The first phase uses the previous state estimate, the current system input, and the system model linearized about the previous state to determine an estimate of the filter state, $\hat{x}_{k|k-1}$ and $P_{k|k-1}$.

The second phase uses the new measurement data to improve the filter state estimation. The useful output from this second phase is the estimation of the current system state. Most condition monitoring applications that use the EKF algorithm use the measurement residual, the difference between output states and measured states, as the health indicator. A larger residual is an indication of deviation from the specified system performance, and therefore an indication of degraded health. The EKF algorithm requires that the equations be linearized, and so at each time step the Jacobian of the system non-linear equations must be calculated before generating the estimate of filter states. An [25], uses the measurement residual of EKF to some success.

Zavarehi [26], makes use of the EKF for observing the valve orifice area, and then predicting the instantaneous fluid flow rate through a hydraulic valve. The work proposes that fluid flow rate into and out of a hydraulic component is a good indicator of component health.

The EKF linearizes the control model about each operating point and is dependant on the current measurement of system states for its output; these make it unsuitable as a predictor. Finally, the EKF algorithm is computationally expensive to run, making it unsuitable for use on the limited resources hardly found in embedded systems.

Observers

Other parameter estimation techniques rely on models to predict future state values. If the observer output states reconstruct measurements of the process, then the observer creates analytical redundancy of those states. Typically, a diagnostic observer generates output states that are indicative of faults, whereas the state observers generate data needed for control. Control observers also tend to operate within a closed-loop environment, whereas a diagnostic observer tends to operate in an open-loop configuration. This requires the diagnostic observer to be more complete, or more robust when considering model uncertainties.

Usually, a diagnostic observer creates a residual that is compared to a threshold to indicate component health. An observer-based residual generator should be designed to compensate for the process input signal, effects of disturbances, and model uncertainty. Non-linear processes cannot usually be represented by linear models, because they often do not operate about a fixed point. If a linear model based residual is used for a nonlinear process, then after a fault has occurred, the model would likely compound modelling errors and exceed its valid range. Non-linear processes can often be modelled by a set of non-linear open loop equation.

The health decision is usually based on comparing the residual to a threshold. Determining the appropriate threshold for error detection is a difficult task; a threshold set too high will make the system insensitive to faults, and setting it too low will result in a high false alarm rate. Setting thresholds has been addressed through statistical data processing, correlation, pattern recognition and also adaptive thresholds. An adaptive threshold dependant on the system input could reduce the false alarm rate. A survey of observers and examples of their use is given in Frank [1].

Model Selection

While the EKF has been shown effective at producing both error signals and noise corrections, including in non-linear cases, it was not selected for this study. First, the EKF relies on having a very good model of the system, and then linearizes the model about the operating point. For the EKF to be most effective, it must be run at very short time intervals, and it does not predict well when the interval increases. Second, uncertainties in the non-linearities lead to uncompensated errors in the state estimation. Third, the EKF is a state-compensator more than a state predictor; it requires state measurement from the most current time to output an estimate of the current state. Finally, the algorithm is too computationally expensive to run on minimum hardware at the desired frequency.

Using non-linear modelling elements is more computationally efficient and more accurate than using higher order linear models for the same task.

The typical observer based approach relies on making small time steps within a system model to try to determine the system's states. Like the EKF, observers often use the current measurements to estimate a set of system states. However, an observer may also be run in a manner which allows it to determine predictions of future states. The reduced-order observer approximates the system by simplifying the linear system model to a lower order.

2.5 *Diagnosis*

The detection and treatment aspects are limited by the granularity of their systems, and the diagnosis mechanism is also dependent on the granularity of the fault detection system, to a lesser degree. A system with a finer granularity simplifies the diagnosis mechanism, as there are fewer system parameters and system states involved in the diagnosis engine for each component. In the simplest diagnosis case, sensors are added to the system, which directly indicate specific faults.

Conditional Statements

Isermann [23], discusses the use of **conditional statements**, which are *if <condition> then <conclusion>* logic statements, to make decisions based on information about system states and parameters from an error detection engine. For some systems, the relationships between faults, events, and symptoms are known to the designer. Then the inference engine can use Event-Tree Analysis, in which it progresses from the known symptoms to the faults that can cause them. The condition part (*if <condition>*) contains facts, represented by symptoms as inputs. The conclusion part includes either events or faults, which are logical causes of the condition.

Fuzzy Logic

Fuzzy logic diagnostics are sometimes used as an extension of the conditional statements or knowledge-based expert systems. In these diagnostic systems, the magnitude of each measured quantity and a rule-based membership function describes a relationship between symptoms and causes. This style of engine is capable of ranking potential causes based on the membership functions and selecting a most probable cause.

Another way to use fuzzy logic is shown in Mechefske [27], which investigates which fuzzy membership function would work best to represent the frequency spectra of various fault conditions for a set of rolling element bearings. Here, the frequency data from a test can be entered into the fuzzy logic filter, and the output is then a set of memberships in each of the fault condition spectra. Each fault condition spectra is tested, and the system detects which condition has the highest membership, thus representing a fault diagnosis.

Artificial Neural Networks

Artificial neural networks (ANN) are frequently used as pattern recognition engines, but are essentially statistical processes, where each parameter within the network is generated by training the ANN to produce different outputs for each failure condition and for the normal operating condition. One network classifier, useful for recognition of patterns, is the Kohonen self-organizing feature map. This type of neural network allows the designer to incrementally extend the domain of patterns to be recognized. The output of the network is a symptom code, which indicates whether the module has identified a possible problem. The Kohonen self-organizing feature map was used in Vingerhoeds [28], to classify engine faults on Boeing 737 aircraft engines.

Neural networks tend to be run off-line; they often require a large history of processed data, and specific test conditions. Karpenko [29], shows how using two tests and the data generated can allow a neural network to detect and identify three different fault types in a pneumatic control valve.

2.6 Summary

This chapter has provided a summary of work related to detecting system faults and monitoring system health, particularly for a marine steering system. It explained the basic mechanisms used for achieving steering in pleasure boats, described the potential modes of failure for a hydraulic system, and described symptoms that may indicate failure or impending failure. The relationship between faults, errors, and failures was introduced.

This chapter introduced the key concepts involved in the design of fault-tolerant systems, and the current approaches taken by other researchers. Health awareness was introduced in the categories of condition monitoring, parameter estimation and state estimation. Finally, the chapter described the existing techniques used in the diagnosis of faults.

Chapter 3: Development of a Distributed Health Monitoring System

3.1 Introduction

This chapter describes the Health Monitoring Layer, and each of the generalized indicators within it. It shows the development of the model based state observer that is the core of the health vector, and shows how it can be used as a redundant analytical sensor.

The Health Monitoring Layer is built upon a previous work [11], which provides a fault-tolerance framework. This chapter begins with an introduction to the fault-tolerant framework to be used as a basis for the health monitoring system. It includes a description of the virtual environment that represents the boat steering system and manages sensor and actuator redundancy.

3.2 Fault-Tolerant Framework

A fault-tolerant system relies on having redundant means of achieving a particular system service. Each system service can be thought of as a causal system that has a set of input states, parameters, output states, and relations between each of those states. Redundancy is needed to maintain any high level service when that service depends on generating a defined output from a given set of inputs even if a sub-component fails. Input states are replicated through analytical or physical redundancy; the relationship between input and output, the actuators or plant, are replicated only through physical redundancy. The commands that operate at the high level input can also be replicated, but to make use of these redundancies in a digital system, the system must be developed with knowledge of the redundant system components.

An important consideration when developing this architecture is the capabilities of the embedded computer. The embedded computer selected operates at only 200MHz, and is described in section 4.2.

This section (3.2) describes the existing fault-tolerant framework that was designed by Bouvier [11]. The health monitoring system is dependent on the concepts and system architecture that was designed for that thesis to provide the virtual boat environment and fault redundancy management.

3.2.1 Object Oriented Programming

Object oriented programming techniques are very appropriate for use in this development environment. Each product, component, subcomponent, etc., can be described by a software class whose attributes and methods provide an abstraction of that component. Where components are replicated, each class can be instantiated as a software object, whose attributes describe that particular component. The inheritance property, which allows a class to by defined as a child of a previously designed class, causes the child class to include all of the attributes and methods of the parent. Furthermore, the polymorphism property allows each child class to re-define attributes and methods that the parent provides, but only in the scope of that child class. This property is particularly useful when designing for diverse redundancy, where each replication has similar attributes, or may require more attributes, and has similar methods, which need to be slightly altered for each case.

3.2.2 System Architecture

Within the fault-tolerant framework, the software methods are categorized as either atomic methods, mid-level methods or high-level methods. A representation of this architecture is shown in Figure 3.1. The architecture shown here is representative of one Electronic Control Unit (ECU).

Atomic methods describe the actual instantiation of components available to the system. Each atomic method has a one-to-one relationship with an input component (a sensor) or an output component (an actuator). Atomic methods make up the system description and the hardware abstraction layer. The hardware abstraction is described in section 3.2.3, concerning the virtual boat steering system.

The mid-level methods make up the redundancy manager. These methods provide the error detection, masking, and error response functionality. They bind to, and aggregate, the atomic methods. The redundancy management layer is described in section 3.2.4.

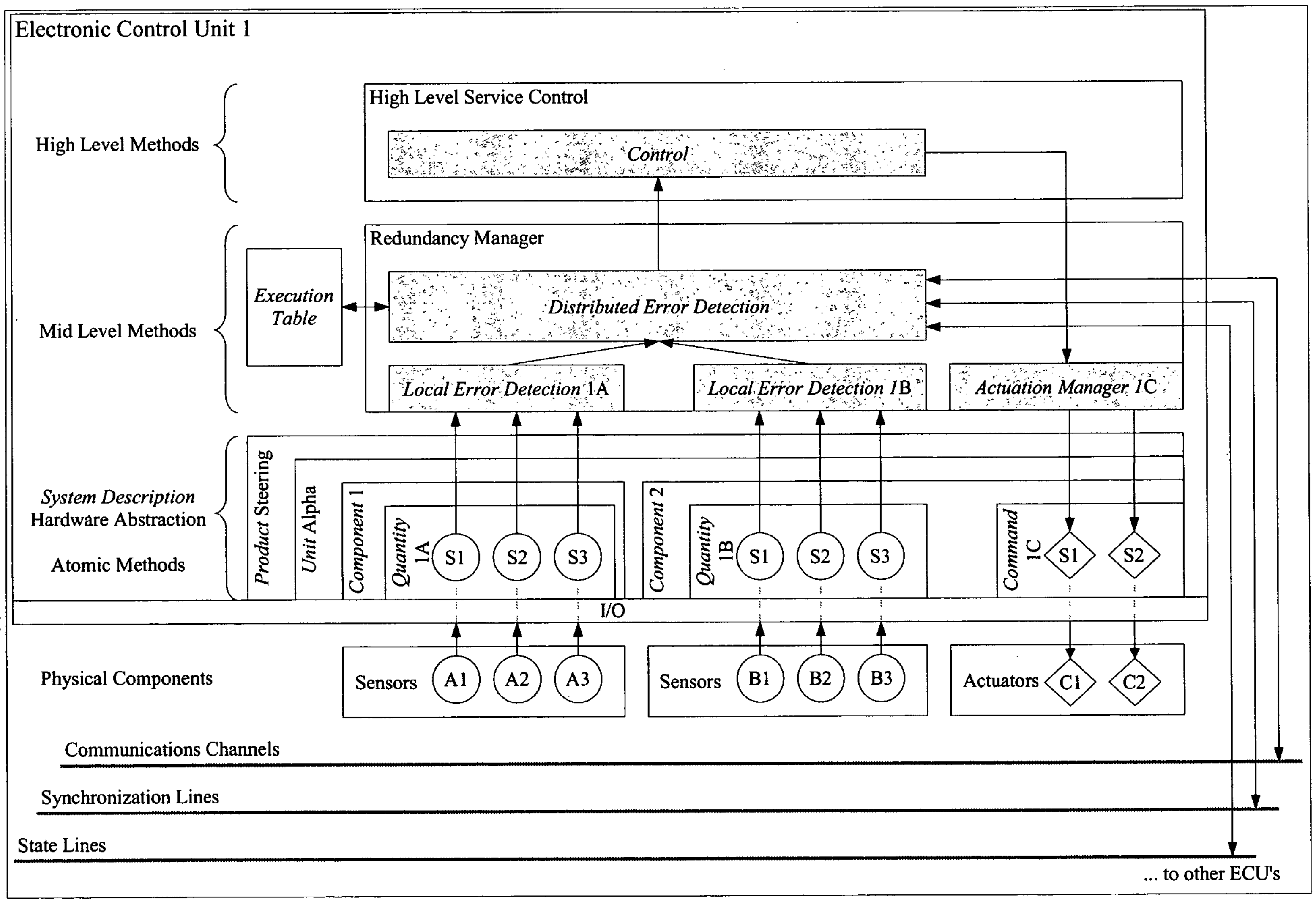


Figure 3.1: Local Fault-Tolerant System Architecture

The high level methods implement the top level services that make up the required functionality that is apparent to an external viewer. These services constitute all of the functionality that is bound to fault-tolerance; their inputs and outputs rely solely on fault-tolerant methods and data that has been checked against errors. They use only the mid-level methods. Consequently, the high level functions organize the system control algorithm, and call upon the redundancy management functions. These features are described in section 3.2.5.

3.2.3 Virtual Boat System Description

In the fault-tolerant steer-by-wire system, the framework is made aware of the hardware by an abstraction of the system description. The virtual boat Hardware Abstraction is shown in Figure 3.2; the names and description of it match the system description object model. The entire system is encapsulated by a top level class, and then divided into four levels.

First, consider the top level class, *Product*. The *Product* class encapsulates a single high level service of the system, which for the current example is a boat steering system. The high level service that the fault-tolerant steering system provides is the link from operator input to steering output. This involves the acquisition of the operator input at the helm, the transfer of the command signal to the actuator, and the delivery of a motion on the rudder or outboard motor to alter the boat's vector. This hardware abstraction is designed to describe the system and its related components, so the *Product* class aggregates its member *Units*, or components. The key units that are required to provide the boat steering mechanism are the helm and the drive, which in this steering system is hydraulic.

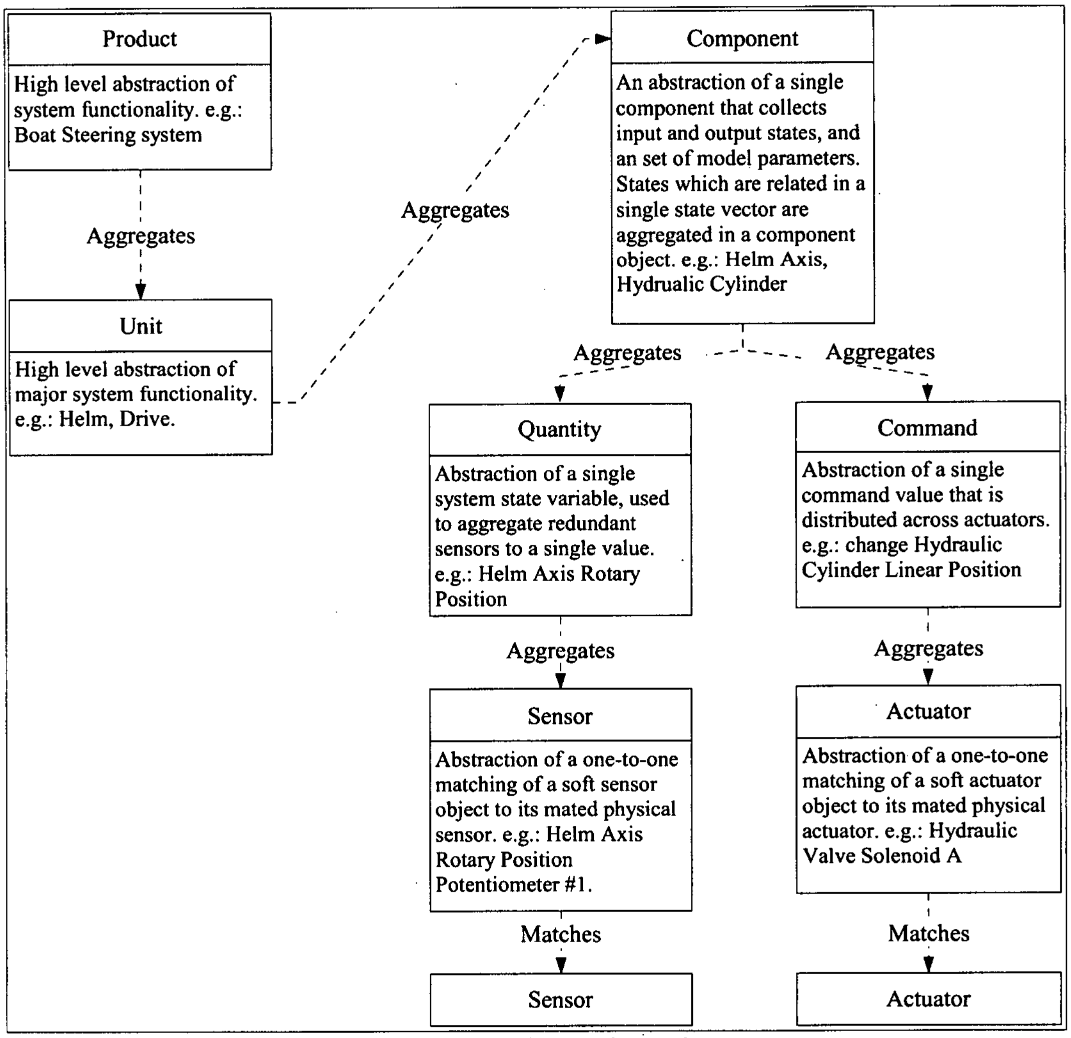

Product
High level abstraction of system functionality. e.g.: Boat Steering system
Aggregates
Unit
High level abstraction of major system functionality. e.g.: Helm, Drive.
Aggregates
Component
An abstraction of a single component that collects input and output states, and an set of model parameters. States which are related in a single state vector are aggregated in a component object. e.g.: Helm Axis, Hydrualic Cylinder
Aggregates
Aggregates
Quantity
Abstraction of a single system state variable, used to aggregate redundant sensors to a single value. e.g.: Helm Axis Rotary Position
Command
Abstraction of a single command value that is distributed across actuators. e.g.: change Hydraulic Cylinder Linear Position
Aggregates
Aggregates
Sensor
Abstraction of a one-to-one matching of a soft sensor object to its mated physical sensor. e.g.: Helm Axis Rotary Position Potentiometer #1.
Actuator
Abstraction of a one-to-one matching of a soft actuator object to its mated physical actuator. e.g.: Hydraulic Valve Solenoid A
Matches
Matches
Sensor
Actuator

Figure 3.2: Hardware Abstraction

The *Unit* abstraction layer is an organizational level, and represents an ideal location to implement a system model of the subcomponents. In the context of the steering system, there are two *Units*, the Helm and the Drive. The Helm *Unit* needs to be aware of all of its state variables, including current position, input torque, and feedback torque. The Helm's function is to provide a platform for the measuring of operator input and providing tactile feedback (tactile feedback was not implemented for this thesis). Fault-tolerance requires that the Helm be able to measure the operator input given the failure of a sub-component or sensor. Each *Unit* aggregates its member *Components*, which creates a state vector representation of the data required by *Unit* to provide its service and implement fault-tolerance.

The *Components* abstraction layer is the organization level where the fault-tolerance management is located. This layer collects all of the input and output states, and a set of system parameters that describe the component. This collection is analogous to a system state space vector. The system inputs are aggregated as *Quantities*, and the system outputs are aggregated as *Commands*. In the context of the Hydraulic *Unit*, there are three *Components* that are known to the system for input or output: the Pump, Valve, and Cylinder. The output states of interest for the pump are the outlet pressure and flow rate, and the input state is voltage. For the valve, the important states are its position, and the voltage control to each of its two solenoids. The cylinder's tracked states are high side and low side pressures, linear position, velocity and acceleration.

The *Quantity* and *Command* abstractions have the same functional level. A *Quantity* is the abstraction of a single state input variable, and is used to aggregate the redundant sensors that can be used to acquire this state. For example, in the Helm *Unit*, the Helm Axis *Component* has the *Quantity*, Helm Axis Rotary Position (HARP). This describes only a single state value, but could be acquired from multiple sensors. A *Command* is the abstraction of a single state output variable or command value that could be issued to multiple identical redundant or interdependent actuators. For example, in the Hydraulic *Unit*, the Valve *Component* has the *Commands*, Solenoid A Power, and Solenoid B Power. In this case, neither actuation command is redundant, but their functions are dependant on each other.

The lowest layer of abstraction then is the *Sensors* level and *Actuators* level. At this level, specific classes exist to form the unary binding of a soft system state variable to a physical device. At the Helm, the *Quantity*, HARP is the collection of the HARP Potentiometer #1, HARP Potentiometer #2, and the integration of the Helm Axis Rotary Velocity (HARV) *Quantity*, which is acquired from the HARV Tacho-generator #1, and from the derivative of the HARP *Quantity*. At the Hydraulic Drive, the *Commands*, Solenoid A Power and Solenoid B Power are bound to objects that, when instantiated, directly indicate Hydraulic Valve Solenoid A and Hydraulic Valve Solenoid B.

3.2.4 Virtual Boat Fault Redundancy Management

The previous section discussed the system description classes, but these classes are part of a larger framework that includes both the local fault-tolerant functionality and the distributed error detection. This subsection discusses how the fault-tolerant classes integrate with the virtual boat environment.

Local Error Management

The Virtual Boat Fault-Tolerant Abstraction of the error management system is shown in Figure 3.3; the names and description of it correspond to the fault-tolerant classes. The *Error Detection* and *Actuation Manager* classes are responsible for local error detection, response, and compensation. These fault-tolerance management classes run at the loop closing frequency of the control system, and the methods that provide the fault-tolerance services operate independent of the system's internal error status. The System Description classes are shown in order to indicate their integration within the Fault-Tolerant architecture, and to provide clarity of the resolution of fault-tolerance provided by this architecture.

The *Local Error Detection* class aggregates all of the sensors known to a given *Quantity*, and uses the information about each *Sensor* to implement an inexact voting scheme, which in turn provides an agreed upon value that is returned to the *Quantity*. It also allows the detection of erroneous sensor data, and the ability to immediately isolate and mask a single sensor fault. Finally, the Local Error Detection (LED) outputs a set of error status indicators for each of the sensors; this data is in turn used by the *Distributed Error Detection* (DED) utilities, as discussed in the following subsection. The *Actuation Manager* class aggregates each of the *Actuators* known to a given *Command*, and sends a value to each of them based on a current *Command* value.

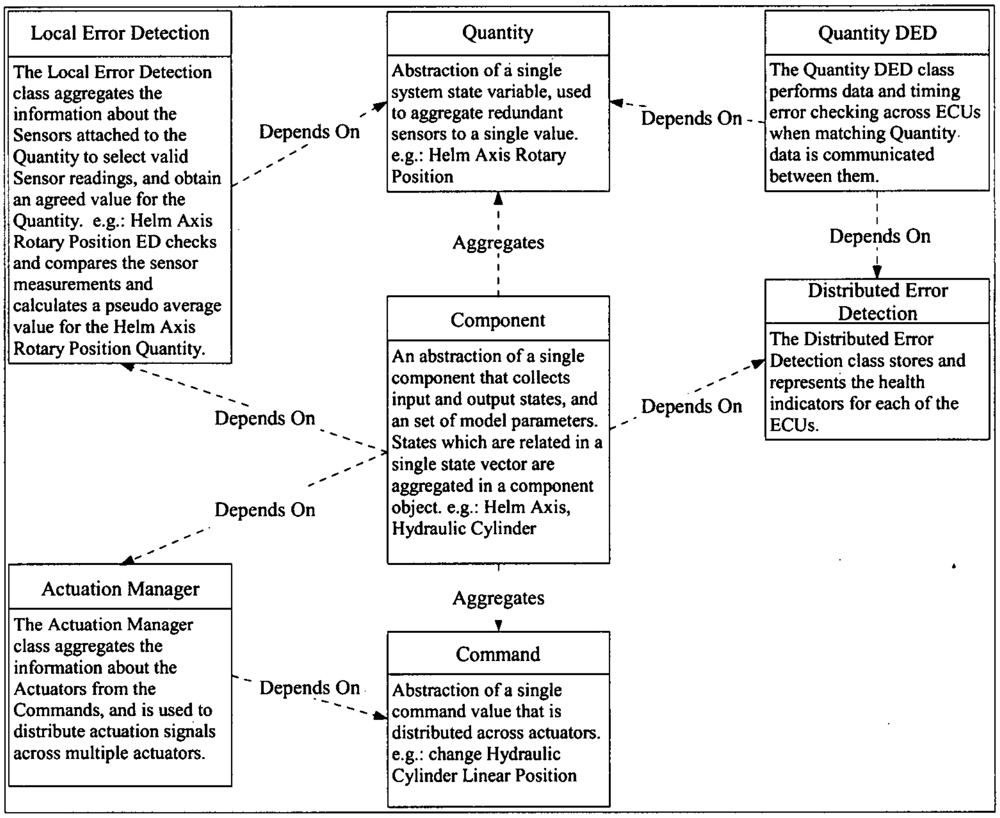

Local Error Detection
The Local Error Detection class aggregates the information about the Sensors attached to the Quantity to select valid Sensor readings, and obtain an agreed value for the Quantity. e.g.: Helm Axis Rotary Position ED checks and compares the sensor measurements and calculates a pseudo average value for the Helm Axis Rotary Position Quantity.
Quantity
Abstraction of a single system state variable, used to aggregate redundant sensors to a single value. e.g.: Helm Axis Rotary Position
Quantity DED
The Quantity DED class performs data and timing error checking across ECUs when matching Quantity data is communicated between them.
Depends On
Depends On
Aggregates
Depends On
Distributed Error Detection
The Distributed Error Detection class stores and represents the health indicators for each of the ECUs.
Component
An abstraction of a single component that collects input and output states, and an set of model parameters. States which are related in a single state vector are aggregated in a component object. e.g.: Helm Axis, Hydraulic Cylinder
Depends On
Depends On
Depends On
Actuation Manager
The Actuation Manager class aggregates the information about the Actuators from the Commands, and is used to distribute actuation signals across multiple actuators.
Aggregates
Command
Abstraction of a single command value that is distributed across actuators. e.g.: change Hydraulic Cylinder Linear Position
Depends On

Figure 3.3: Fault-Tolerant Framework

The class diagram shows how the *Local Error Detection* and Actuation *Manager* classes are related to the *System Description* classes. The redundancy management classes need to be aware of the *Component*, *Quantity*, and *Command*, that they are attached to. The services of the *Local Error Detection* class are invoked by the *Detect Errors* method. This method first gathers the individual sensor measurements and performs the executable assertions (the *E_A:* method) on those measurements. The executable assertions are the first set of error detection routines run for a sensor; they ensure that each data point is within the suitable range of the sensor, and that its variation based on its previous value is acceptable. Then, data replication assertions are performed across the sensors, which forms the second part of the error detection process, and will detect Byzantine faults. Each sensor value is compared to the rest to determine if they are within an accuracy tolerance of each other; this will identify any sensor of giving data which is not similar to the other sensors. Next, any sensor detected as faulty is removed from the current operations, and an indicator flag is raised. Finally, the sensors that are considered error-free are subject to an inexact voting algorithm, which calculates a weighted average value. This output value is returned and becomes the *Quantity* state value.

The services of the *Actuation Manager* class are invoked by the *Actuate* method. This method uses the *Command* state value, and computes output values for each of the attached member *Actuators*. Then, each actuation command value is sent to the appropriate Actuator output channel.

It should now be evident that the System Description classes only provide information about the presence, relationship, and state values of each component in the system, and whose methods must be aggregated into management functions. The redundancy management classes, the control classes, and communications classes all use the data and methods from the System Description classes to implement the desired system services.

Distributed Error Management

The *Distributed Error Detection* representation is shown in Figure 3.3. The Distributed Error Detection classes manage ECU redundancy, and provide the error detection utilities for the value and time domains. They output the error status of each of the ECUs as calculated locally, a system-wide agreed state value for each of the *Quantities*[3] aggregated within the *QuantityDED* class, and an error status indicator for each of the *Quantities* that are checked.

As a mid-level object, a *QuantityDED* object outputs a value for its *Quantity* that the distributed system agrees upon, and an error vector describing the fault status of each of the ECUs which contribute to that *Quantity*. The *QuantityDED* checks the values from each ECU to assert that the sensors provide similar data at each localization. It also compares the error status reported by the *Local Error Detection* object at each ECU to verify agreement on sensor health.

If the QuantityDED finds a discrepancy in error status from a *Local Error Detection* object, it will flag the detection of an ECU fault at the desired fault granularity. For example, if a system has multiple *QuantityDED* objects, one of the *QuantityDED* objects may detect a fault. In this case, the detection of the fault could result in the ECU being marked as faulty, or the *QuantityDED* object may mark the *Local Error Detection* object of that specific ECU as faulty.

3 Note: A *Quantity* is the name to the system abstraction that represents a single system state. The abstraction is required as a single system state may be determined by multiple methods. Analytical and physical redundant sensors that output the same state are aggregated by a *Quantity*.

3.2.5 High Level Service Layer

As discussed in section 3.2.2, high level methods make up the services that are required for the system to provide the system specification performance. In particular, the loop closing and the execution table are considered high level methods.

Execution Table

The execution table allows for the system to change which methods are used in real-time, thereby adapting to faults that may affect the software code. This behaviour is desirable when an ECU error is detected, and can ensure that communication with the unit is suspended, and an alternate distributed error detection mechanism is used.

A step is the abstraction of a task to be accomplished by the software. Any task might be completed by multiple methods, each method providing diverse redundancy of similar functionality. The step aggregates the redundant methods and their timings. For example: a single step might call the functions responsible for data acquisition, fault-tolerance, communication (sending and/or receiving), control, or actuation.

The execution table class collects all of the required steps. Each step is designed to be run at a particular time, as communication, data acquisition, and actuation are all time-dependant events. A lookup table is used to select which redundant methods are used from each step, allowing for dynamic reconfiguration when an error is detected that affects the ECUs. The execution table is a high level object, and is called within a loop run by the global function.

Control

The controller classes aggregate the virtual descriptions of the quantities, commands, model, and algorithm, and are not directly dependent on any physical component. The controller object methods are called from within the execution table, making this a high level object.

3.3 *Health Monitoring Approach*

Section 2.4 discusses a number of approaches to achieving health awareness: condition monitoring, parameter identification, and state estimation. In general, the condition monitoring approach involves adding physical sensors to measure a direct indicator of health. Alternatively, parameter identification and state estimation use model-based approaches, and sensors that may already exist for control purposes, to create a measure of health.

Condition monitoring will often require the creation or addition of new sensors to measure signals which directly indicate system performance. These sensors are specific to each application, and each recorded signal is usually indicative of a specific fault or operating condition. Condition monitoring systems are programmed with awareness of a number of fault conditions, and a diagnosis mechanism compares the system state with the condition signals to provide a diagnosis of system health.

Model-based approaches are preferred for this project because they offer the opportunity to find health indicators from the sensors that already exist to provide control, and so they use the hardware redundancy that already exists. Generally, model-based approaches allow for two choices of how to proceed. The first method is parameter estimation, and the alternative is state estimation.

The most significant difficulty with parameter estimation occurs when the system model is non-linear. Typically, parameter identification approaches rely on the least squares algorithm to determine the parameters, and this algorithm requires the model to be not only constant, but also linear.

Most real mechanical systems cannot be accurately represented by a single linear model. Usually, mechanical systems have a number of non-linearities, from saturations, to relays, to friction. Moreover, a number of mechanical systems are not time-invariant; they change not only with their internal parameters, but also with time. To accurately model such system, typically non-linear model elements and multiple model states are required.

The hydraulic steering system used as an example in this thesis has a number of non-linearities that manifest themselves as saturations, dead-bands, rate-limiters, relays, and transport delays. The actual system model used is discussed in detail in Chapter 4, section 4.3.

In this thesis, the modelling strategy uses a second-order system model to estimate system dynamics. The linear model follows a set of non-linear models that are executed prior to the second-order model. The combination non-linear to linear model allows for reasonably accurate output to be achieved from a straightforward and computationally efficient algorithm. It also allows the non-linearities and the linear portion of the model to be uncoupled which provides the benefit of running the model only once to acquire all health indicators.

3.3.1 Model Selection

The use of a second-order model has a number of benefits. In particular, reducing the system to its dominant second order model allows its performance to be specified in terms of gain, damping ratio, and natural frequency. Then gain, damping ratio and natural frequency can be used to set dynamic thresholds on acceptable system behaviour, from which deviation indicates health degradation. This model also has the benefit of providing an analytical predictor of system state, which can be used as a redundant sensor when one can assume that the input states to the observer are error-free. First-order models do not allow the modelling of resonance, nor do they provide for the inclusion of complex solutions. The second-order model is the lowest order model available that allows resonance to be modelled.

This model only needs reasonable accuracy over a short period of time, or it can even be incomplete. It assumes that the natural frequency, gain, and damping ratio are affected by deteriorated health, and that the common errors will be observable by changes in one of these parameters.

Finally, it is expected that the modelling elements might be used in series. A single process could be constructed from a number of non-linearities and second-order transfer-functions. Alternatively, a number of small processes or system components could be individually modelled using the non-linear plus second-order transfer function, and then all of the models could be cascaded in series to produce an overall model of the system which has many non-linearities and multiple second-order transfer functions. The overall model could then be equivalent to a higher order model.

3.3.2 Indicator Selection

One of the advantages of using a model-based approach is the opportunity to re-use system sensors included for control in the models that power the health indicators. One indicator of health is the current system state in relation to the system reference state; to obtain this measure (the control error signal), models are not necessarily required. This mode, instantaneous error, is called the Instantaneous Response Indicator and is discussed in section 3.3.2.

The instantaneous error is not always a good indicator of system health, nor does it provide enough information to localize component health. The instantaneous error can be small or zero when the actual state crosses the command state (e.g.: a system with constant command signal, but oscillating state). To detect this behaviour, a history of past states is required, and can be indicated by modelling the system dynamics. Indicating the health of the system dynamics involves modelling the system response over a period of time that approximates the system time constant, and comparing the output of the modelled dynamics to the actual system state. This system dynamics error is called the Dynamics Response Indicator, and is discussed in section 3.3.3.

Modelling system dynamics over a period of time will help to indicate that the system behaves as expected over a short period of time. However, because the Dynamics Response Indicator is run over a short period of time, it becomes susceptible to missing health errors over a longer period (e.g.: the system response to a slow moving input, or the average system response compared to the expected response over a time period). Generating the expected response requires a system model to be run over the entire period of monitoring. If the period is significantly longer than the settling time of the dominant dynamics, then only the non-linear part of the model needs be run. This mode, the average state change error, is called the Average Response Indicator and is discussed in section 3.3.4.

3.3.3 Instantaneous Response Indicator

The **Instantaneous Response Indicator (IRI)** is the difference between the current reference state and the actual current system state. This residual value is also typically used by the control system itself. The error signal is scaled to the range of the sensors used, and reduced by an accuracy threshold. For this indicator, a lower absolute value of the indicator indicates a healthy measure for the system, whereas a high absolute value indicates that the system health may have deteriorated. When validating this indicator, the result should fall below a determined threshold value for normal operating conditions. Duration, direction and magnitude of the health indicator are all signs of system malfunction.

In this thesis, the IRI is the commanded input position relative to the position of the cylinder. A high error signal could also indicate that the rate of change in the input signal is faster than the system can accommodate – which can arise from an error state at the helm or input processing, perhaps too high a rate of change in the helm position. Clearly, under normal conditions, one would expect that the steering system can handle the range of motion that the helm is capable of providing. The given example however is for a health vector referring to the cylinder, so this signal is an indicator that the cylinder is unable to track the reference signal.

3.3.4 Dynamics Response Indicator

The system **Dynamics Response Indicator (DRI)** measures the difference between an actual system state and the upper or lower boundary expected of that state. The expected value of the state is found by running a model of the system which includes the non-linear aspects as well as the linear portion of the model. The linear portion of the model is a second-order function, and the prediction is generated over a period that is less than the time constant of the second-order equation.

The upper and lower boundaries of the expected state describe where the system should be assuming boundaries on natural frequency, damping ratio, gain and phase of the system with respect to the input over the prediction period. Deviation from the path boundaries results in a non-zero error. Larger error magnitudes indicate a deviation of actual system dynamics from the expected system dynamics, be it in phase, frequency, or amplitude.

Magnitude and direction of this health indicator are the key elements of health for this sensor. While duration is important, it is more important to examine repeatability of errors with similar inputs, or any error signals that are persistent for more than one time step or recur at the expected natural frequency of the system. During normal operation, the output from this sensor should be near zero; magnitude of the error signal should be small and any variation should appear to be noise.

In the example used for this thesis, the model is used to predict the velocity of the cylinder in response to the commands sent to the pump and valve.

3.3.5 Average Response Indicator

The **Average Response Indicator (ARI)** is the difference between expected change and the measured change over a period of time. The response to a slow moving input is checked over a period of time that is significantly longer than the settling time of the dominant system dynamics. This enables the luxury of not running the oscillating system dynamics model, as its influence would be averaged over the period. However, a non-linear model would be run over the period, and all of the input way-points would be gathered and included in the non-linear portion of the model. Using this method, the average response from the non-linear model to the full history of inputs is compared to the average value of the actual system state's change over the same period.

Direction, magnitude, and duration of any error signal are all important factors to monitor in this signal. When validating this sensor, a signal with long duration indicates a change in static behaviour of the system, which could in turn be used to tune the results of the system Dynamics Response Indicator sensor. Matching error sign with the IRI signal could be a result of increased system lag, and opposite signs can indicate out of phase tracking.

In this thesis' example, this indicator is instantiated as the difference between expected average velocity and actual system average velocity over the time window. A large error signal here indicates that the system is not responding to an input command as expected. A change in performance can be attributed to either the input (actuator fault) being incorrect, or that the model structure or model parameters no longer adequately approximates the actual system (system/process fault).

3.3.6 Model Development

The Health Monitoring System has three available inputs: the system state vector, system reference vector, and control signal vector. Each component of the model is instantiated as an object that behaves as either a discrete transfer function or as a non-linear transfer function.

The model is not intended to be run continuously, where each step is dependant on the previous, but instead only over short intervals and synchronized with the actual system states following each prediction. This principle is shown in Figure 3.4. In this figure, a single threshold predictor is shown. In this version, the model is updated to include all of the inputs, *u(t), u(t+Δ)* ... *u(t+nΔ)*, over the prediction period and the starting states of the system, *y(t)*; it then predicts the system end point, $\hat{y}(t+n\Delta)$, and upper and lower thresholds, $\hat{y}_{ub}(t+n\Delta)$, $\hat{y}_{lb}(t+n\Delta)$. After each prediction, the actual system states are stored by the model, and it predicts the state value at the end of the next forecast period.

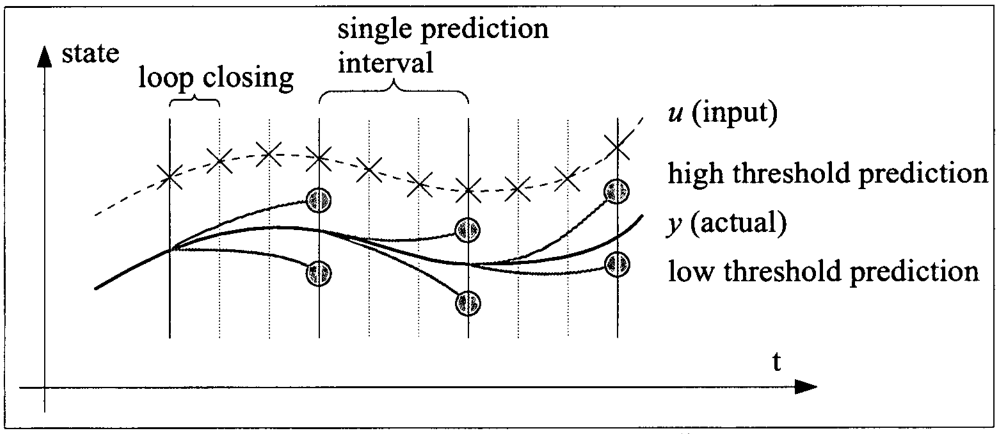


Figure 3.4: Single State Threshold Predictor

There are updates for the input between each of the predictions shown in Figure 3.4. The period of prediction is significantly longer than the loop-closing frequency, and the system health indicators should be updated at each loop closing. Each set of predictions then represents results for a window of system data that moves with time. When a prediction occurs at each loop closing, the moving windows overlap with previous predictions, as shown in Figure 3.5. In this figure, the system input data is known across each model window, but each prediction has a different starting state. Using this technique, the health indicators are updated at each loop closing.

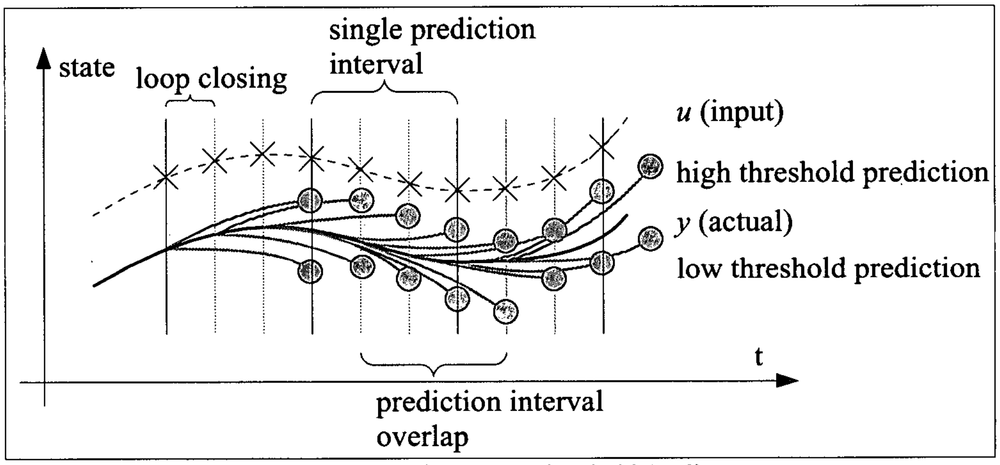

state
loop closing
single prediction interval
u (input)
high threshold prediction
y (actual)
low threshold prediction
t
prediction interval overlap

Figure 3.5: Multiple State Threshold Predictors

Figure 3.6 shows the locus of the predicted thresholds, and also the locus of the forecast of the system state corresponding to the outputs shown in Figure 3.5.

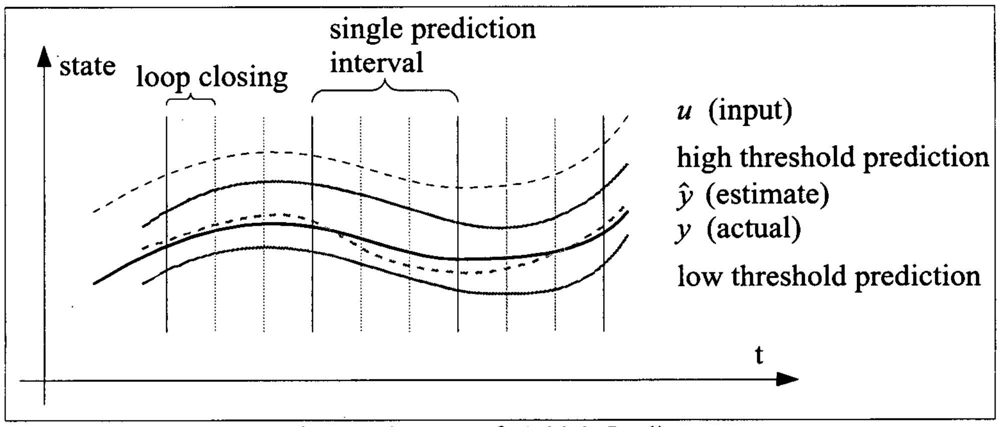


Figure 3.6: Locus of Multiple Predictors

It should now be clear that the system dynamics model is restarted to the actual system states at a regular interval. This synchronization allows the dynamics to be monitored by the DRI, but also makes the indicator unable to track system changes which operate more slowly than the system dynamics. It also prevents it from detecting errors that would arise from the system having a significant offset from the commanded state. In order to compensate for the loss of tracking of slow signals, the ARI was created. The IRI was created to compensate for the inability of the average response indicator and the dynamic response indicator to detect errors arising from the system not adequately tracking the commanded state.

3.4 Dynamic Performance Analysis

3.4.1 Objective

The objective of this algorithm is to use specified system parameters and acceptable limits on those parameters to determine the range of acceptable outputs for that system. The linear part of the system model considered is second order, and has the form:

$$\frac{d^2 y}{dt^2}+\frac{d y}{dt}2\,\omega_n\zeta+y\,\omega_n^2=k\,\omega_n^2 u \tag{3.1}$$

Where:

y is the current system state with respect to time, t;

ω_n is the undamped natural frequency

ζ is the damping ratio

k is the system gain

u is a general input function of time, t

The system is assumed to maintain constant values of ω_n and ζ over the duration of the estimation period and during normal operation; they are constant in each record. The final algorithm implementation supports parameter updating at irregular intervals as desired in code. These values are known and specified by the system designer. Furthermore, a range of acceptable values of ω_n and ζ are specified such that $\omega_{lb} \leq \omega_n \leq \omega_{ub}$ and $\zeta_{lb} \leq \zeta \leq \zeta_{ub}$ during operation. ω_{lb} is the lower boundary of ω_n and ω_{ub} is the upper boundary of ω_n. Likewise, ζ_{lb} is the lower boundary of ζ and ζ_{ub} is the upper boundary of ζ. The problem then becomes finding the maximum and minimum solutions to Equation 3.1 when solved for y.

The timing interval selected is long enough for the effects of dynamics to be seen. The timing interval is selected to match approximately 0.8 radians at the undamped natural frequency. The actual sampling interval and loop closing time of the system is on the order of one tenth of the system time constant, which is significantly faster than the forecasting period of the model. The long forecast period and the multiple loop closings in between can be used to provide a series of predictions over the actual forecast time, with each prediction made from a previously known system state.

To maximize the potential modelling benefit of having a history of inputs, a number of modelling techniques were investigated. Techniques tested include analytical solutions assuming a step input, ramp input, and parabolic input, and numerical integration techniques including Predictor-Corrector, Runge-Kutta, and a Modified Euler integration. The use of embedded systems minimizes processing capabilities; the use of analytical equations was investigated, but processing capability proved to be a constraint to their use, as using numerical analysis to solve the equations is a processor intensive task.

The limit on processing capabilities leads one to select a solution where the calculations performed at each loop closing use a minimum of processor time. The preferred algorithms have most of their calculations performed off-line (by the designer) or during a configuration phase (where the control loop is not running). This limitation leads to the selection of a solution that is expressed in a solved algebraic form.

A number of options were considered for determining the best way to calculate an algebraic solution to the equation. The input function was replaced by step, ramp, best fit ramp, parabolic, and cubic spline functions, each varying which input points to use. After comparing the output from each of the solutions, the Analytical Solver for a Parabolic Input was selected. In this case, three of the input points are used: the start point, end point, and mid point.

3.4.2 Analytical Solver for a Parabolic Input

With the parabolic input, starting from Equation 3.1, found in the previous subsection, and using a parabolic input of the specific form:

$$u(t)=\left\{a_1 t^2+a_2 t+a_3 \;\middle|\; a_1=\frac{2(u_0-2u_1+u_2)}{\Delta^2},\, a_2=\frac{-3u_0-4u_1+u_2}{\Delta},\, a_3=u_0\right\} \tag{3.2}$$

Where:

Δ is a discrete time interval representing half the forecasting period

$u_0=u(t)$, $u_1=u(t+\Delta)$, and $u_2=u(t+2\Delta)$

The specific solution then is found to be:

$$
\begin{aligned}
y(t)= \quad & \frac{1}{\Delta^2\omega^2}\Bigg[-4ku_0+8ku_1-4ku_2+16ku_0\zeta^2-32ku_1\zeta^2+16ku_2\zeta^2 \\
& -8ktu_0\zeta\omega+16ktu_1\zeta\omega-8ktu_2\zeta\omega+6ku_0\Delta\zeta\omega-8ku_1\Delta\zeta\omega+2ku_2\Delta\zeta\omega+2kt^2u_0\omega^2 \\
& -4kt^2u_1\omega^2+2kt^2u_2\omega^2-3ktu_0\Delta\omega^2+4ktu_1\Delta\omega^2-ktu_2\Delta\omega^2+ku_0\Delta^2\omega^2 \\
& \frac{-1}{2\sqrt{-1+\zeta^2}}\Bigg(E^{t(-\zeta\omega+\sqrt{-1+\zeta^2}\omega)}\Delta^2\omega\Bigg(-dy_0-k\frac{(8u_0\zeta-16u_1\zeta+8u_2\zeta+3u_0\Delta\omega-4u_1\Delta\omega+u_2\Delta\omega)}{\Delta^2\omega} \\
& +\frac{(\zeta+\sqrt{-1+\zeta^2})(-y_0\Delta^2\omega^2+k(u_0(-4+16\zeta^2+6\Delta\zeta\omega+\Delta^2\omega^2)+2(-4u_1(-1+4\zeta^2+\Delta\zeta\omega)+u_2(-2+8\zeta^2+\Delta\zeta\omega))))}{\Delta^2\omega}\Bigg)\Bigg) \\
& +E^{t(-\zeta\omega-\sqrt{-1+\zeta^2}\omega)}\Delta^2\omega^2\Bigg(y_0-\frac{k(u_0(-4+16\zeta^2+6\Delta\zeta\omega+\Delta^2\omega^2)+2(-4u_1(-1+4\zeta^2+\Delta\zeta\omega)+u_2(-2+8\zeta^2+\Delta\zeta\omega)))}{\Delta^2\omega^2} \\
& +\frac{-dy_0-\frac{k(8u_0\zeta-16u_1\zeta+8u_2\zeta+3u_0\Delta\omega-4u_1\Delta\omega+u_2\Delta\omega)}{\Delta^2\omega}}{2\sqrt{-1+\zeta^2}\,\omega} \\
& +\frac{(\zeta+\sqrt{-1+\zeta^2})(-y_0\Delta^2\omega^2+k(u_0(-4+16\zeta^2+6\Delta\zeta\omega+\Delta^2\omega^2)+2(-4u_1(-1+4\zeta^2+\Delta\zeta\omega)+u_2(-2+8\zeta^2+\Delta\zeta\omega))))}{2\sqrt{-1+\zeta^2}\,\Delta^2\omega^2}\Bigg)\Bigg]
\end{aligned}
\tag{3.3}
$$

This solution can be manipulated to provide two Equations of the form:

$$
y(t+2\Delta)=A_0y(t)+A_1\frac{dy(t)}{dt}+A_2u(t)+A_3u(t+\Delta)+A_4u(t+2\Delta)
\tag{3.4}
$$

One equation is valid for $\zeta < 1$, and the other valid over the range for $\zeta > 1$. The equations of this form are used in the algorithm to produce the best prediction of y, as well as the predictions for the maximum and minimum limits of y at the extremes of allowed values for ω_n and ζ. The development of equation 3.4 and final values for A_0 through A_5 are shown in Appendix B.

The problem to solve is then to find the maximum and minimum $y(t+2\Delta)$, where only the parameters ω_n and ζ change. The solution to the maximum/minimum problem then is as follows:

$$
\frac{\partial y}{\partial \omega_n}=0 \qquad \frac{\partial y}{\partial \zeta}=0
\tag{3.5}
$$

In the current equation forms for $y(t)$ however, no analytical solution to these equations could be found. To eliminate the exponential functions from the solution for

Equation 3.5, the exponential functions of Equation 3.3 were expanded as power series to four terms, which yields the function:

$$\begin{aligned} y_{j+2\Delta} = {} & y_j + t\,dy_j - t^2 \frac{\omega_n}{6}\left(6\,dy_j \zeta + \left(k\left(-4u_{j+\Delta} + u_{j+2\Delta}\right) + 3y_j\right)\omega_n\right) \\ & - t^3 \frac{\omega_n^2}{6}\left(dy_j - 4\,dy_j \zeta^2 + 2\left(k u_j - y_j\right)\zeta\,\omega_n\right) \end{aligned} \quad 3.6$$

The power series expansion shown in Equation 3.6 is accurate to within 1% of the analytical value at up to 0.8 radians. The maximum or minimum condition for this equation is found at:

$$\omega_n = \frac{\sqrt{\frac{dy_j}{t}\left(-2\,dy_j t - 3k u_j + 8k u_{j+\Delta} - 2k u_{j+2\Delta} - 3y_j\right)}}{-k u_j + y_j} \quad 3.7$$

$$\zeta = \frac{3\,dy_j + k\,t\,u_j \omega_n^2 - t\,y_j \omega_n^2}{4\,dy_j\,t\,\omega_n} \quad 3.8$$

3.4.3 Results

Algorithm comparison was initially performed in Matlab. Selected results are shown for the Analytical Solver for a Parabolic Integrator. The results were gathered using a normalized second order system with gain set to 1. The first selected input form is a sine wave, operating with an amplitude of 1.0, and a frequency which is half of the natural frequency of the system being analyzed.

The system envelope of the parabolic integrator, in response to the sine wave input is shown in Figure 3.7. The figure shows five data series. The first series is the input signal to the transfer function of the model and to the prototype Dynamics Response Indicator. The three series that track with a phase lag are the actual system output, and the high and low boundaries of system performance as predicted by the Dynamics Response Indicator. The final data series is actually at zero for this entire test, but it is representative of the error between the actual system output and the boundaries, and is non-zero only when the actual output deviates outside of the high and low limits.

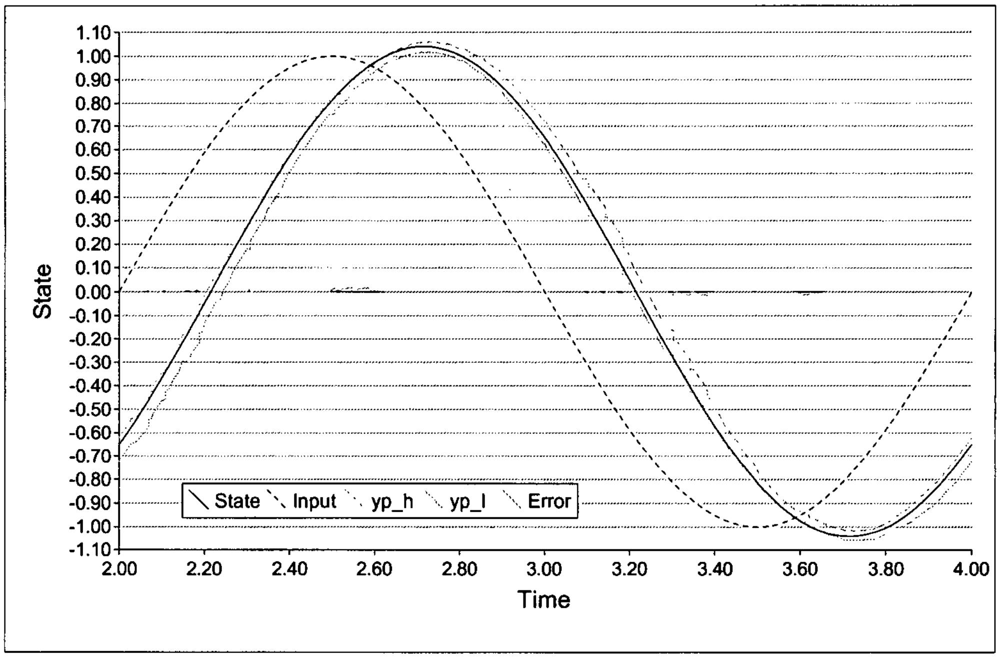


Figure 3.7: Parabolic Analytical Integrator Response to Sine Wave Input

In this test, the parameters ω_n and ζ are allowed a tolerance of 20% from their idealized values. It is apparent that for this test, this system performance envelope entirely encapsulates the output of the system. The parabolic approximation is fairly accurate across the entire range of the sine wave input signal.

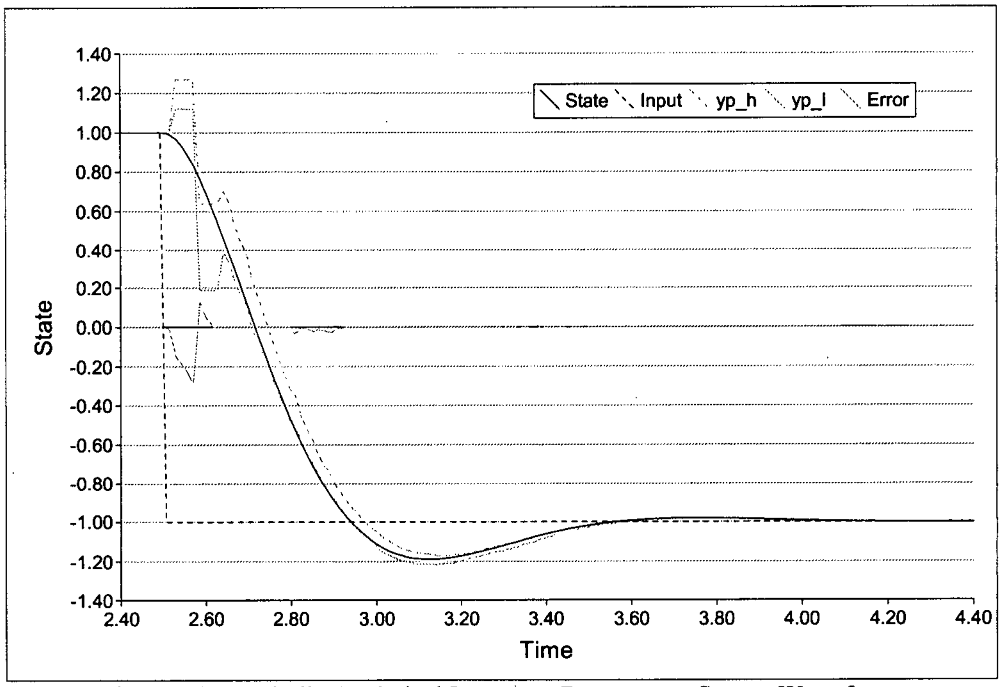


Figure 3.8: Parabolic Analytical Integrator Response to Square Wave Input

The second test shows system response to a square wave input in Figure 3.7. This chart has five data series. The first series is the square wave input. There is the series which represents actual system output, and then the high and low boundaries of system performance as predicted by the Dynamics Response Indicator. The final series is representative of the error between the actual system output and the boundaries.

At the beginning of the step, the high and low boundaries increase, where the actual system drops. This occurs because of the parabolic approximation of the input, whose three data points are 0, 0, and -1, which will result in a parabola with a maximum value greater than 0. The high and low boundaries then quickly drop to below the actual system, when the data points in the parabolic input are 0, -1, and -1. It is clear now that the parabolic approximation is adequate when the system input is a continuous function, but performs poorly when the input is discontinuous.

A second set of errors is shown in the interval of 2.8 to 2.9 seconds, and is a limitation of the method used. This error type led to the use of the prediction for maximum and minimum predictions given in equations 3.7 and 3.8. It is also important to note that the power-series expansion is accurate to 1% at 0.8 radians of the undamped natural frequency; this indicates that small deviations in the system from the thresholds is expected, and the magnitude of the error in this interval is approximately 1%.

3.5 *Health Monitoring Layer*

The desired output from a health monitoring layer is a set of values that are indicative of the current health status for a given component. Each unique health monitor should aggregate a set of indicators, and all of the methods that describe those indicators. It should be designed to be aware of the error status of its input states. The health monitoring layer should collect each of the local monitors to create a composite health vector that describes the health of components and subcomponent at the desired resolution.

The Error Detection classes are responsible for detection of errors with the sensors, and determining an agreed value for a particular quantity. Furthermore, they are also responsible for disabling faulty sensors. The health indicators are used to monitor actuator and plant performance, with that data being used in a diagnosis engine, which in turn would trigger the Actuation Management classes. The diagnosis phase of the Health Monitoring System is not included in this analysis.

The Health Monitoring indicators are model based, as described in Section 3.3, which makes each indicator dependent on the current system state and the input applied over a period. The Health Monitoring Layer is necessarily reliant on the systems' sensors, observable states, and controller inputs. Therefore, to achieve its goals, it needs to use the system hardware abstraction, and also to use the current error status of all of the state variables used.

The performance measuring objective of the Health Monitoring System led to the selection of a scheme where each indicator returns a quantitative measure corresponding to the degree of deviation from good behaviour. The indicator's scalar value increases as the performance error increases.

3.5.1 Layer Framework

Figure 3.9 shows a flow chart of the data used by the health monitor system during normal operation. During the first step, the *RunModel* method is used, which updates the model elements, and generates a prediction for the system state, and the upper and lower boundaries for the analytical sensor. If a sensor has already failed, the Redundant Analytical Sensor, shown as 1.(B) is executed, allowing the Local Error Detection object to properly check for sensor agreement.

The second step in the data flow for the Health Monitoring system is updating the system states with the values produced by the Distributed Error Detection routine. This then allows the calculation of the output values for each of the Health Indicators.

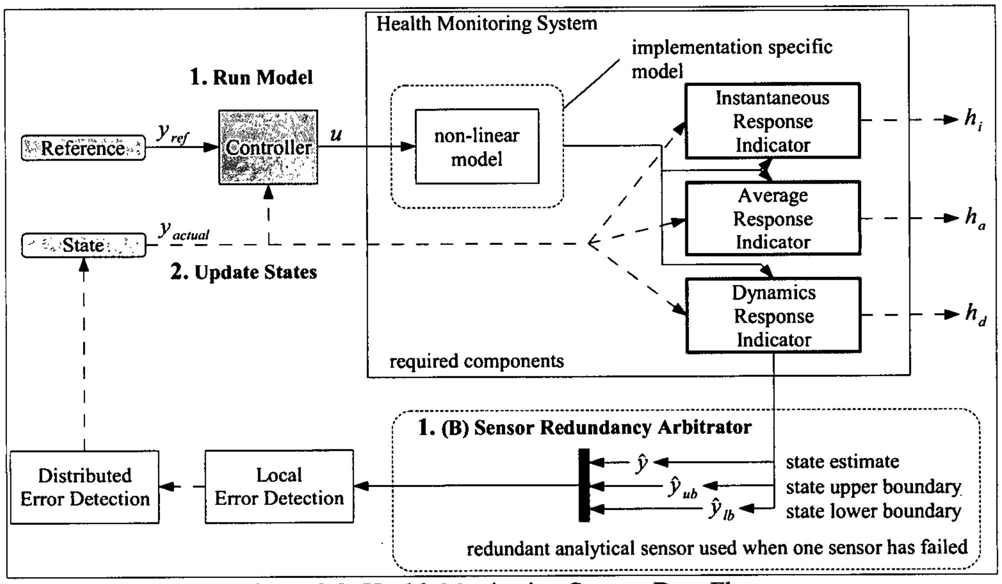


Figure 3.9: Health Monitoring System Data Flow

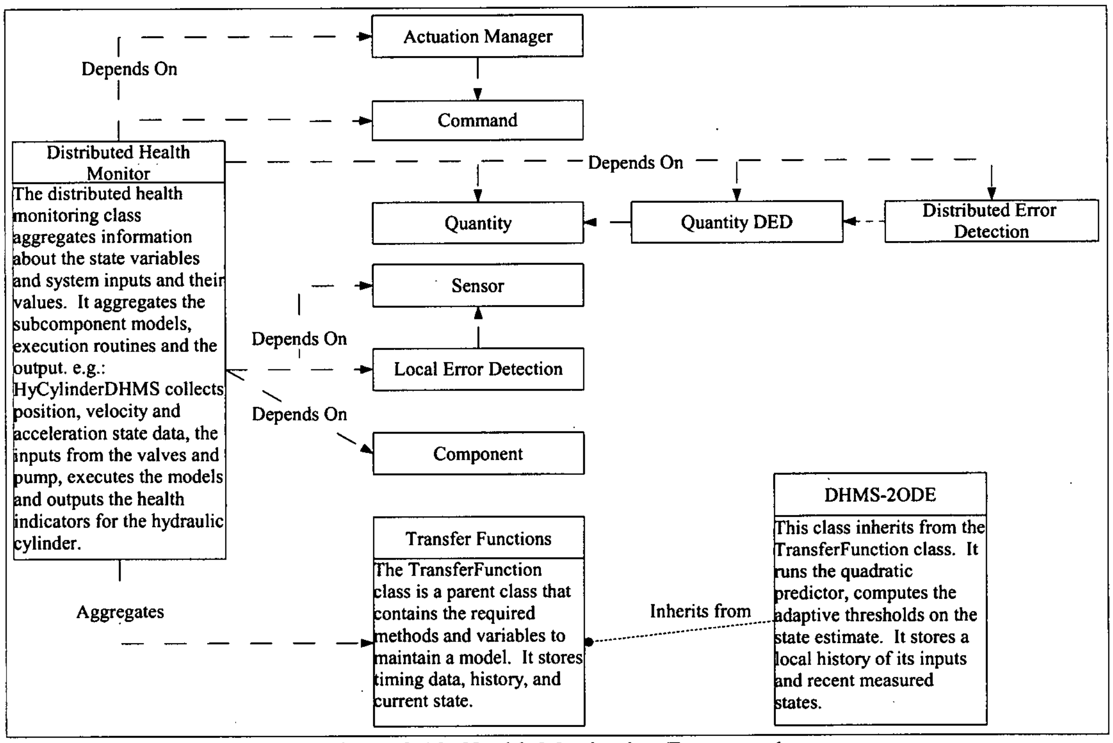

Actuation Manager
Depends On
Command
Distributed Health Monitor
The distributed health monitoring class aggregates information about the state variables and system inputs and their values. It aggregates the subcomponent models, execution routines and the output. e.g.: HyCylinderDHMS collects position, velocity and acceleration state data, the inputs from the valves and pump, executes the models and outputs the health indicators for the hydraulic cylinder.
Depends On
Quantity
Quantity DED
Distributed Error Detection
Sensor
Depends On
Local Error Detection
Depends On
Component
Transfer Functions
The TransferFunction class is a parent class that contains the required methods and variables to maintain a model. It stores timing data, history, and current state.
DHMS-2ODE
This class inherits from the TransferFunction class. It runs the quadratic predictor, computes the adaptive thresholds on the state estimate. It stores a local history of its inputs and recent measured states.
Aggregates
Inherits from

Figure 3.10: Health Monitoring Framework

The abstraction of the framework for the Health Monitoring Layer is shown in Figure 3.10. Detailed descriptions of the classes are given in Appendix C. The health monitoring class is responsible for organizing the indicator output data and parameters. Objects of this class are updated at an integer multiple of the loop closing frequency, and provide health information for the given component if the input state variables are error-free. The system description and fault-tolerant framework classes are shown to indicate how they relate to the Health Monitor.

Each of the modelling objects that make up the health monitor model is collected in the Health Monitoring Class. This allows the system to manipulate its execution rate, to select individual models for execution, to access model data, to display current status, and to adjust each model at run time. All of the models are run by the *RunModel* method, where the *RunModel* method calls the equivalent method from each of the models within the Health Monitor collection; as an alternative, the function could call a Forth word to run the models in a particular order. The model output data is then stored, and it can be compared to the system state vector to generate the Health Indicators, which in turn would be used by a diagnosis object.

The DHMS-2ODE class provides the second order threshold estimator and state predictor described in section 3.4. It is also the key element in the Dynamic Response Indicator. When the *RunModel* method is called, the model predicts the system output and the upper and lower thresholds of the output state. The *UpdateStates* method is used to store the actual current system state. These two functions are separated and called independently so that the prediction of the output state and the thresholds can be calculated and used as an analytically redundant sensor in a Local Error Detection object when one of the redundant physical sensors has failed. The LED object uses the predicted state and thresholds in the inexact voting scheme, which determines the actual current system state.

Input states are normally used from the Distributed Error Detection object. Typically, control signals are collected directly from the commands that are sent to the actuators. The health monitoring object can be attached to any type of component; it could represent a *Unit*, *Component*, *Command* or a specific *Actuator*. This gives the benefit of allowing the designer to set the granularity of the Health Monitoring System to any superset of the granularity defined by the System Description objects.

3.5.2 Model Framework

All of the models used are based on the TransferF class. The models take advantage of the inheritance and polymorphism properties of the object oriented Forth extension. Each of the models used is registered in the Health Monitoring object, allowing for some modelling parameters to be changed system-wide, and on demand.

For normal operation, the designer specifies how the models are run. This function is implemented as part of the instantiation, and is registered in the health monitoring object to be used with the *RunModel* method. Some of the model classes are introduced here. Please see Appendix C for additional software documentation.

The *Saturation* class is an example of one of the non-linear models that make up the health monitoring predictor. It allows for double sided saturation at two values. The non-linear models make use of the *RunModel* method by default, and the *UpdateStates* method will only update the internal state, not the output state.

The *RateLimiter* class is designed to check the velocity over a period of time and if it exceeds a given maximum speed, definable in each direction, truncates that speed to the limit. Like the Saturation class, the *RunModel* method is used to add the new input and update the system model. The *UpdateStates* method can update an input, but will not update the output.

The *DelayRelay* class is a special case of non-linear models. It tracks an internal position state that activates output at the set thresholds. This class is used to describe the behaviour of the solenoid operated valve and the dynamic delays that it applies to the hydraulic system.

The *TimeTolerance* class provides a tolerance to uncertainty of measurement timings, by allowing for a state value to be compared to threshold values at different times. In particular, it uses the *u_h* and *u_l* variables, which are high and low thresholds, and checks the *y* state to see that it lies between those thresholds, not only at the current time, but also within a time tolerance into previous recorded history or also for subsequent values.

The *TransDelay* class provides a transport delay on the input signal equal to the delay parameter. The *UpdateStates* method allows the history of states to be overwritten without affecting the current output, while the *RunModel* method drops the earliest history point to the output state and adds the current input to the queue.

The *WindowAvg* class computes the average value of a state over a given period of time. The averaging window moves with each update, subtracting the earliest value and adding the newest to a running sum when the *RunModel* method is used. The *UpdateStates* method allows the user to overwrite the current sum.

3.6 Summary

This chapter has discussed the fault-tolerant framework that the health monitoring functions are designed to interface with. It also discusses the three health indicators that are used, and the models used to describe them.

It shows how the analytical solution was derived for the prediction of system state, adaptive thresholds of prediction, and the Dynamics Response Indicator. The chapter also shows the preliminary results achieved that led to the selection and the development of the analytical model used.

Finally, this chapter discusses the health monitoring object, and how it is dependant on the classes that describe the system, and the error detection classes. It discusses the algorithm used. The non-linear models that were developed for the project are also described.

Chapter 4: Validation and Test of Health Indicators

4.1 Introduction

In order to use a Health Monitoring scheme, appropriate indicators must be selected, tested, and results validated. This chapter describes the design of a hydraulic steering system, the steer-by-wire components and the instantiation of the health monitoring vector components. Overall system performance, and also the testing and validation of its results are discussed in detail. The performance of the health monitoring system is demonstrated with respect to the motion of the cylinder. The success of the indicator is dependant on the health of many other components: the health monitor models the cylinder's performance with respect to commands sent to the valve and pump motor.

The first part of this chapter provides a detailed description of the steer-by-wire system, the design of the health monitoring models, and the instantiation of the health indicators. The second part (of this chapter) describes the experiments used to demonstrate performance of the indicators, and their results.

4.2 ***Design of the Hydraulic Steering System***

The experimental system models the components expected to be present in the hydraulic power steering system of an outboard driven power boat. The test bed includes all required hydraulic components from the pump to the cylinder. For the purpose of isolating the test of the Health Monitoring system, the helm input is computer generated.

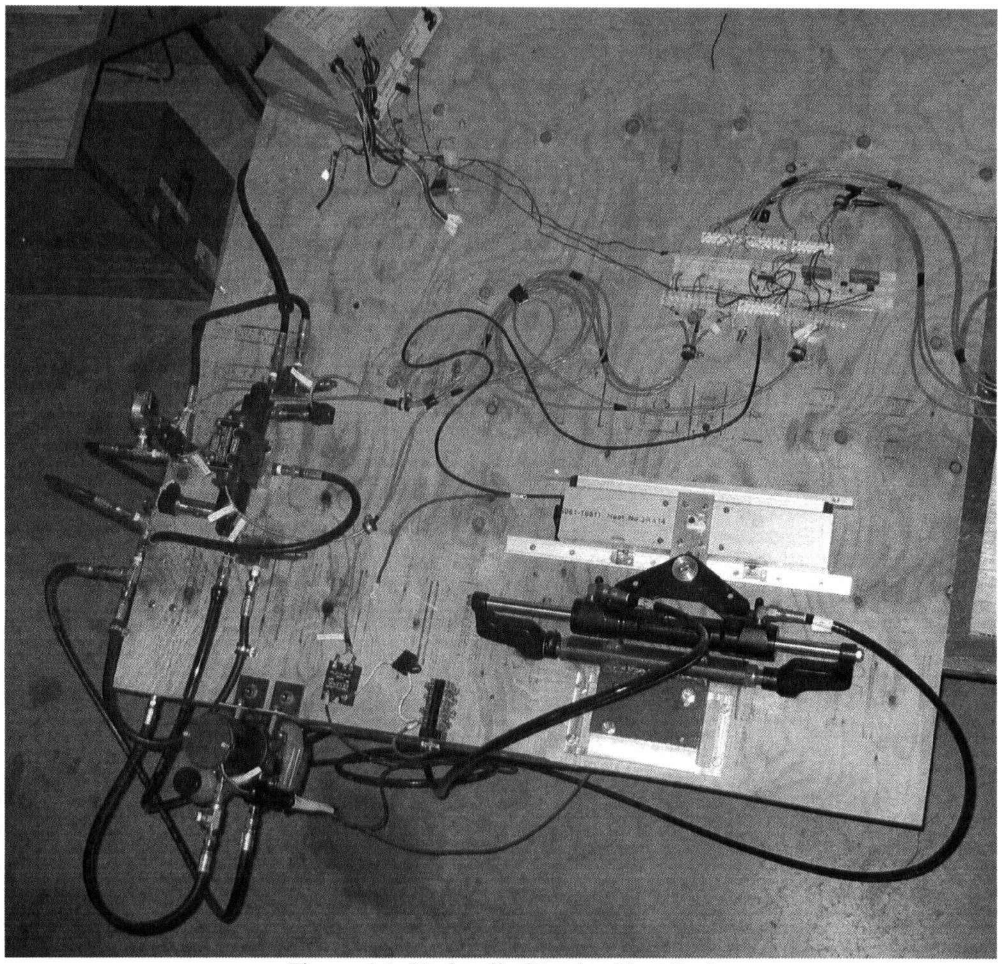


Figure 4.1: Hydraulic Steering Test Bed

Under normal operating conditions, it is expected that the drive-by-wire steering system would be controlled by multiple computers. The distributed computing environment would include error checking and data sharing via network communications, as is implied in section 3.2.4. To isolate and test the health monitoring functions, the experiments done here involved only one ECU without communication with the others in the network.

A photograph of the hydraulic steering apparatus is shown in Figure 4.1, and the electronic components are shown in Figure 4.2. The majority of the embedded computer configuration was completed by Bouvier [11]; the focus of the work described in this thesis is the control of the hydraulic system.

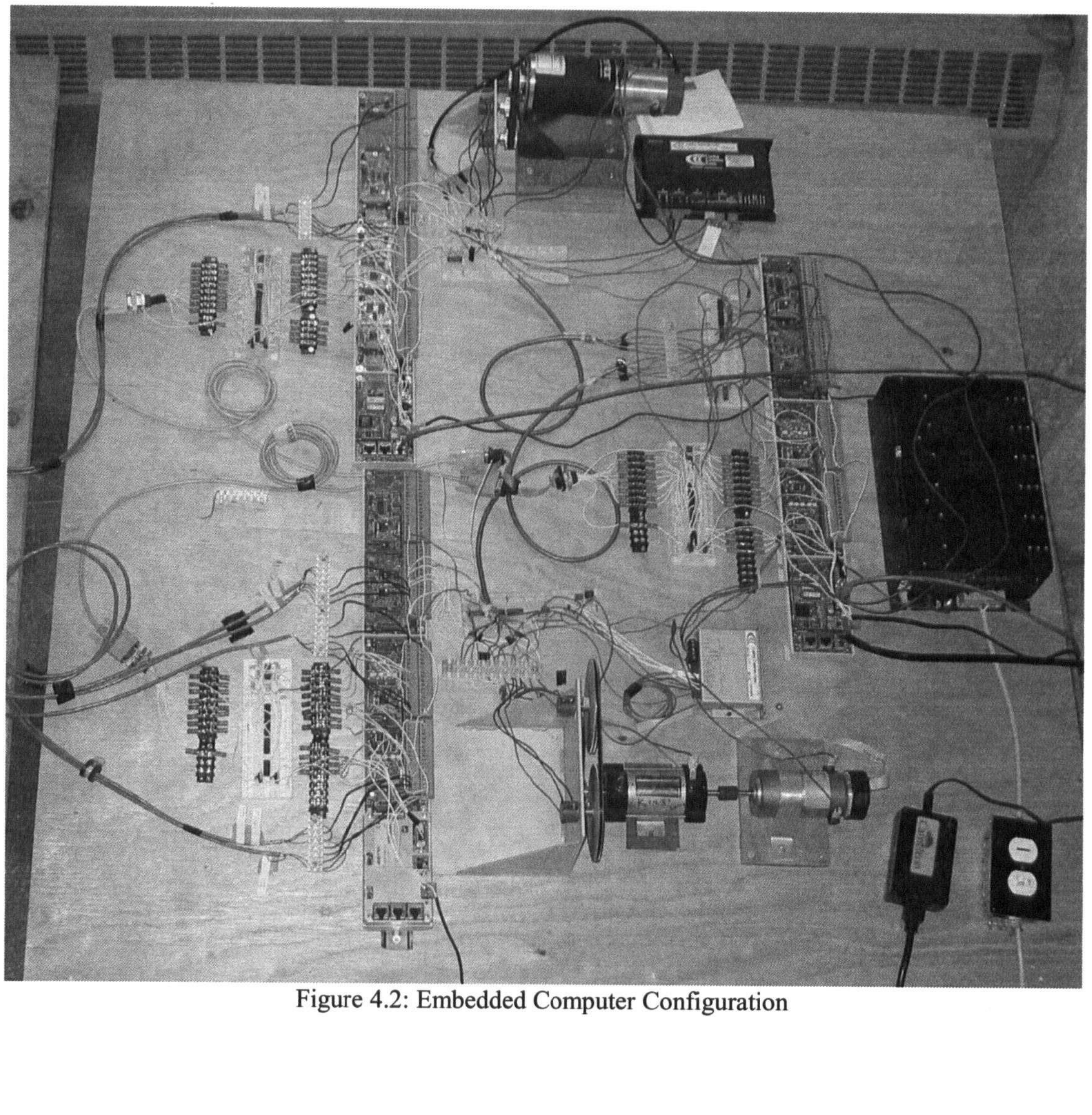


Figure 4.2: Embedded Computer Configuration

4.2.1 Hydraulic Circuit

The circuit diagram for the hydraulic actuation system is shown in Figure 4.3. This circuit diagram shows the configuration of the hydraulic components, and builds on the diagram shown in Figure 2.4. The parts lists are shown in Table A.1 and Table A.2.

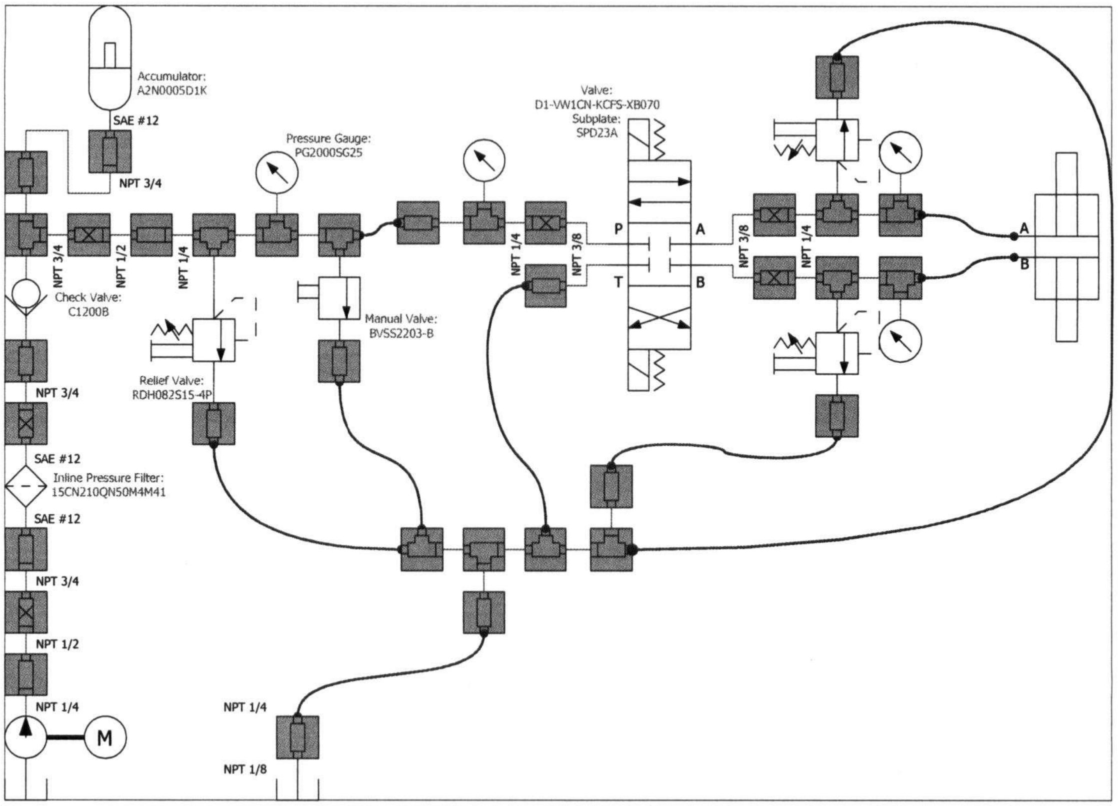


Figure 4.3: Hydraulic Circuit Diagram

Motor: The pump, motor, and reservoir are supplied as a combined part by Teleflex, and are representative of components that would be used in a power-steering application. The motor is a typical brushed DC motor, and is powered by a 24 V DC supply from a large diesel truck battery. The power supply to the motor is switch controlled by the embedded computer via a Crydom solid-state relay. The switch used in this application could be replaced by an amplifier, which would allow for control of the pressure or flow rate from the pump. The inductance of the motor is low, which gives it a low electrical time constant, and hence when switching the power on and off to the motor, the current inrush is very high. This makes PWM control of the relay difficult, as current requirements are too high. The power circuit is also fitted with an emergency interrupt switch to cut power manually.

Pump: The pump, included with the motor and reservoir, is a gear pump and is directly powered by the DC motor. The pump is fitted with internal relief valves set to 1400 psi on the up port (A), and 650 psi on the down port (B). The pump is reversible, so both ports are also connected to the reservoir. In combination with the given DC Motor, the pump is rated to provide 200 psi and 175 in^3/min at 24 V and 40 A, and up to 1000 psi and 120 in^3/min at 24 V and 60 A.

Reservoir: The reservoir has a capacity of approximately 1 L. The volume of the entire circuit, including the filter and accumulator is approximately 2.5 L. The circuit was filled with Teleflex "Hynautic" Steering Fluid.

Cylinder: The steering cylinder is a Teleflex SeaStar, model number HC5345. The cylinder contains a double-acting piston with area of 1.0 in^2, and a stroke of 8.0 in; Symmetrical piston areas are required in this form of steering application, where symmetrical steering forces in both directions is important.

Control Valve: The control valve from Parker, is a 4 Way-3 Position valve with solenoid operation and spring return. The neutral position (centered position) of the valve is a 4 Way hold; no fluid flows through any of the four ports. The first activated position directs flow from the pump via Port P to one side the hydraulic cylinder via Port A, and connects Port B, which is the cylinder's low pressure side, to Port T, which is the return to tank line. The opposite active position directs flow from Port P to Port B, and Port A to

Port T, which reverses the direction of flow at the cylinder. The solenoids are controlled by the embedded computer via a solid state relay (powered by a 24 V, 0.96 A holding current). Experiments found the valve to have a switch on time of 50 ms. The spring return time, which is the time it takes to move to the centre position, was measured to be 110 ms. When the opposite solenoid was switched on, the powered return time was found to be 70 ms.

Accumulator: The accumulator is sized to provide some supplementary flow to the system if the pump output decreases, up to 5 inches of stroke. It should also relieve some fluid shock when the system is operating around 700 psi. It should be noted that the maximum pressure observed was 500 psi on the manual gauge.

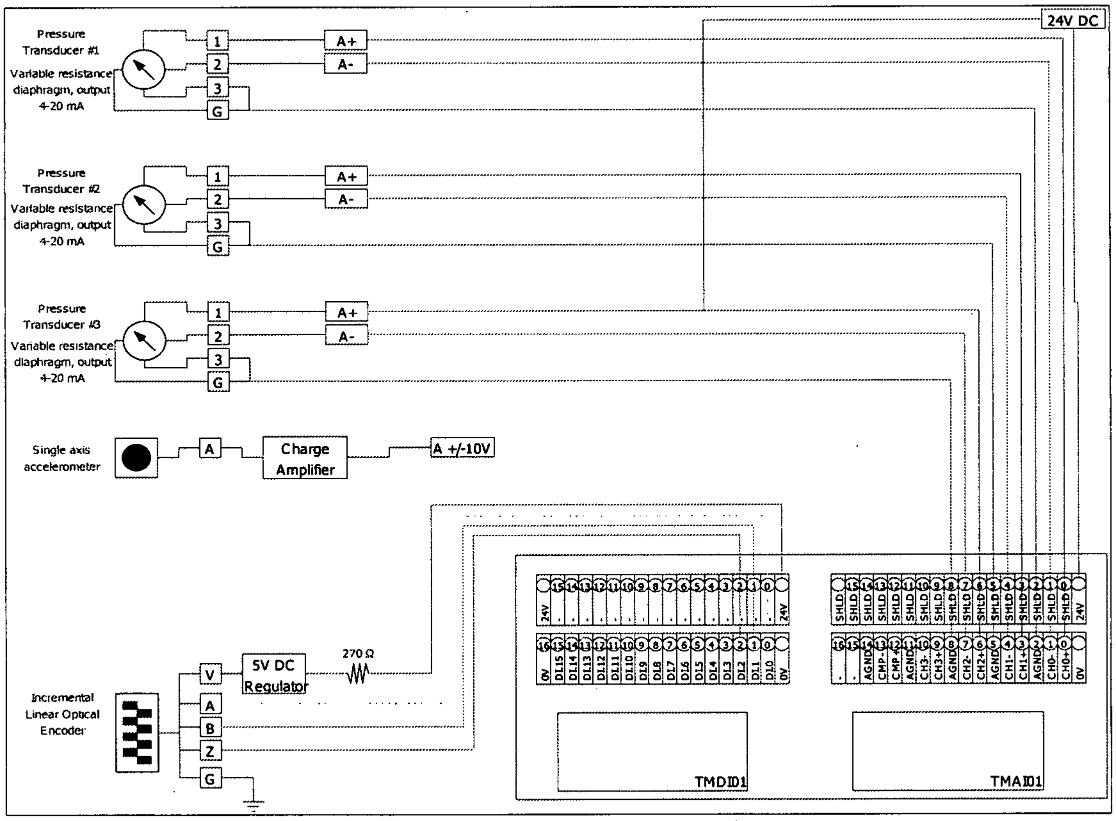
Filter: The Parker in-line pressure filter is selected to provide pressurized contaminant filtering. The filter was selected to accommodate flow rates of up to 175 in^3/min, which resulted in a fairly large port size, which necessitated using a number of fittings to step up the size, and then step it down following the filter.

Manual Valve: The manual valve was installed to aid in filling the circuit, including reducing pressure and flow rate at the cylinder to bleed air from the circuit. Installing this valve also provides a convenient way to drop system pressure for maintenance, and also enables a series of tests to be run simulating a leakage condition with decreased pressure and flow rate.

Pressure Gage: The pressure gauge has a pressure scale from 0 to 2000 psi, and is placed following the filter and check valve, and between the accumulator and the control valve. The gauge is suitable for checking average pressure during operation and the holding pressure that the system achieves.

4.2.2 Electrical Sensor Circuits

The system sensors provide state information about the hydraulic steering system to the embedded computer platform. The system has the capability to sense pressure information from three pressure transducers, and one single axis accelerometer connected to the analog input card as shown in Figure 4.4. The system also detects axis position with the optical linear encoder connected to the digital input card, which is also shown in Figure 4.4.


Pressure Transducer #1
Variable resistance diaphragm, output 4-20 mA
Pressure Transducer #2
Variable resistance diaphragm, output 4-20 mA
Pressure Transducer #3
Variable resistance diaphragm, output 4-20 mA
1
2
3
G
A+
A-
24V DC
Single axis accelerometer
A
Charge Amplifier
A +/-10V
Incremental Linear Optical Encoder
V
A
B
Z
G
5V DC Regulator
270 Ω
24V
DL15 DL14 DL13 DL12 DL11 DL10 DL9 DL8 DL7 DL6 DL5 DL4 DL3 DL2 DL1 DL0
0V
SHLD
AGND
CMP-
CMP+
CH3- CH3+ CH2- CH2+ CH1- CH1+ CH0- CH0+
TMDI01
TMAI01
Figure 4.4: Sensors Electrical Circuit Diagram

The implementation of the Distributed Health Monitor only requires the position data from the linear encoder sensor for the selected health vector. A complete implementation of the Distributed Health Monitor and fault-tolerant system would include the pressure transducers and the accelerometer, however they are not included in this system. The full parts list of electrical components used for the sensor circuit is listed in Table A.4.

4.2.3 Electrical Actuation Circuits

The embedded computer platform interfaces with the hydraulic steering system via the actuation circuit shown in Figure 4.5. The valve solenoids and the pump's motor are activated by the digital output board. The output pins controlling the valve are capable of logical high and low states, switched in software. In addition to logical high and low state control, the output pin controlling the pump's motor is also capable of PWM output in the frequency range of 16 Hz to 3900 Hz. When used with an appropriate amplifier, this PWM output will allow the pump and motor to be operated in a pressure controlled or flow control states. The complete parts list of electrical components used in the actuator circuits is presented in Table Appendix A.4.

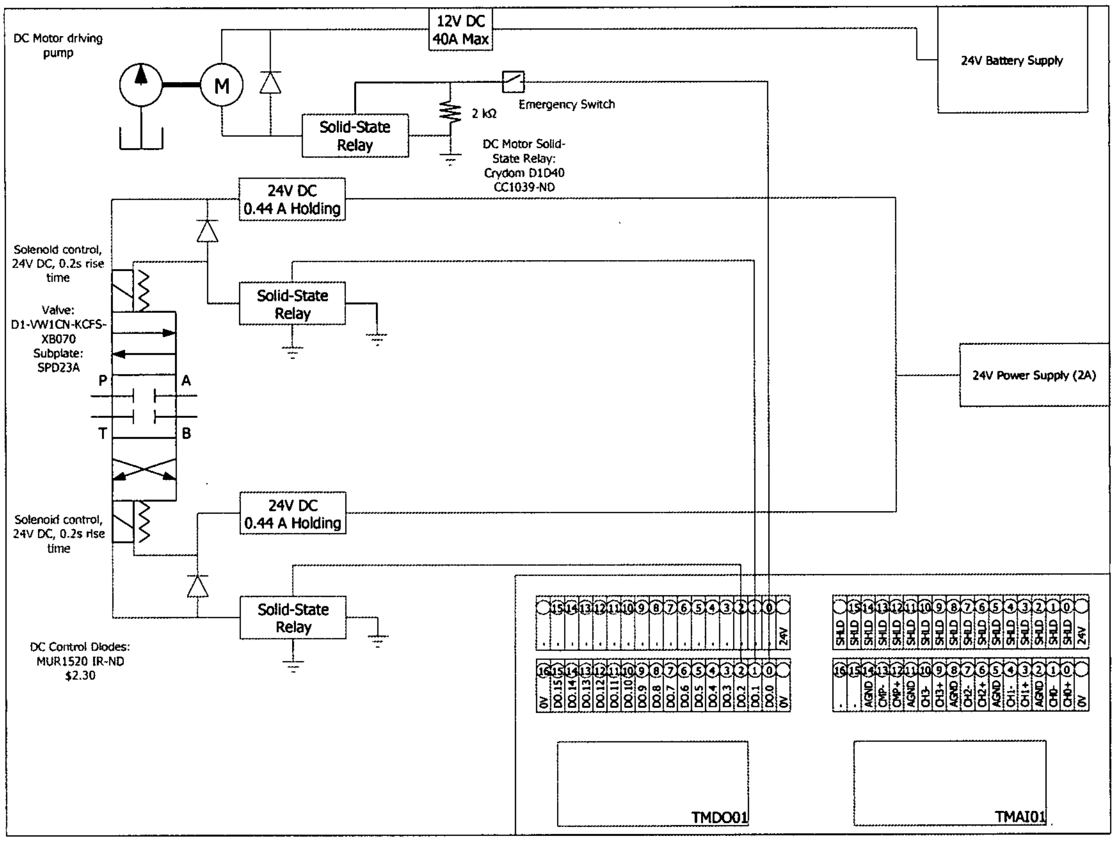


Figure 4.5: Actuators Electrical Circuit Diagram

4.2.4 Embedded Computing Platform

The embedded computing platform is based on the SCM40, manufactured by EXOR International. The SCM40 includes at 64 bit CPU operating at 200 MHz, and an on board programmable FPGA. The CPU is the NEC MIPS VR4131, which uses a RISC architecture, while the Xilinx Spartan IIe FPGA includes 200-600k gates. The SCM40 has 16 MB of RAM, 2 MB of bootstrap flash ROM, and 16 MB of SmartMedia flash ROM. The SCM40 is mounted to a carrier board, with ports for 24 V DC power supply, as well as RS232, RS232C, RS485, field bus, Ethernet 10 Base-T, CAN2.0b, and TPWire support.

Using the TPWire connection, the SCM40's carrier board is connected to two passive carrier boards which are used to house the digital input and output, and the analog input and output cards. Mounted on the first carrier board are the digital output and input cards. The digital output card is the TMDO01 from Sitek. The TMDO01 board includes 16 isolated channels, each capable of 12 bit PWM output, with a maximum delay time of 300 μs. Output current ranges from 10 to 500 mA. With the configured software system, this system provides 4 channels of PWM output, and 12 channels of controlled output. The digital input card is the TMDI01 from Sitek. This board includes 16 channels of digital input, each with impedance of 3 kΩ and maximum input delay of 50 μs.

Mounted on the second carrier board are the analog input and output cards. The analog input card is the TMAI01 from Sitek. The board includes 4 single ended or 8 differential analog input channels. Each programmable channel supports one or two single-ended voltage inputs, one differential voltage input, one current input, one RTD thermometer input, or one thermocouple input. Input voltage ranges are user programmable from the range of +/-10 mV to +/-10 V, and can be unipolar or bipolar. Input current ranges can be selected as 0-20 mA or 4-20 mA. Channel resolution is 12 bits, and conversion rate is 6 μs plus a programmable delay. The analog output card is the TMAO01 from Sitek. This board has 8 output channels of -10 V to 10 V and 12 bit resolution.

4.2.5 Embedded Computing Software Environment

The Steer-by-wire and the health monitor applications are developed in Forth. In order to ensure portability of the application over multiple hardware platforms, the ANS Forth de-facto standard was selected. The object oriented extension used was developed by McKewan, and is ANS Forth compliant. It supports the major object-oriented concepts such as inheritance, polymorphism, and aggregation. Within this object-oriented extension, classes are defined between the delimiting words *:Class* and *;Class*. Inside each class definition, instance variables may be declared as any ANS Forth variable, or as an aggregated class. The scope of these variables is limited to the each object declared, and cannot be accessed directly from other objects of the same type. Also inside each class definition, methods are defined between the delimiting words *:M* and *;M*, and the last character of every method name must be a colon (":"). After defining a class (e.g. *Class1*), an object (e.g. *Object1*) can be instanced by calling the command *Class1 Object1*. It is then possible to access a method within that object (e.g. *Method1:*) by calling the command *Method1: Object1*. The method *ClassInit:* is automatically called when the class is instantiated. The method's definition may be changed, which allows the developer to initialize the object's instance variables to a set of initial values.

4.3 Instantiation of the Dynamic Health Monitoring System

4.3.1 The Cylinder Positioning Model

The form of the non-linear model used is shown in Figure 4.6. The output from the non-linear model is used as input to the Dynamics Response Indicator and the Average Response Indicator, as shown in Figure 4.7 and Figure 4.8, respectively.

As can be seen in Figure 4.6, the output from the controller is a flow rate vector. It is converted into a scalar flow rate command at the pump and a unit vector representing flow direction at the valve. The pump model is simplified as the magnitude of controller command signal, and passed to a relay, thus representing the state of the pump as being on or off. The valve is represented as a relay, where each switching is dependent on the current state of the relay. The state-dependent relay has three possible output states, one, zero and minus one, which constitute the direction of fluid flow through the valve. The time the valve spool takes to make each of these transitions is different. There are six possible transitions, and the timing for each transition is programmable into the state-dependent relay model class.

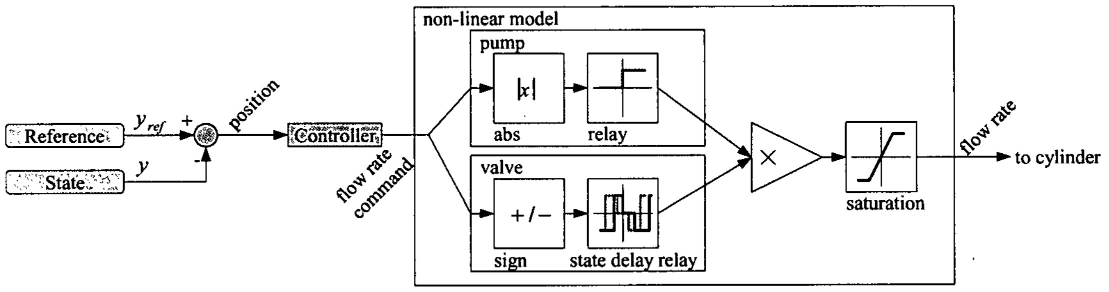


Figure 4.6: Non-Linear Model

Finally, the output from the state-dependent relay and the output from the pump relay are multiplied to yield a directional flow rate that is processed by a double sided saturation block. This block models a saturation of the flow rate in each direction at programmable set-points (not necessarily symmetric). The output from this block is the expected flow rate vector at the cylinder.

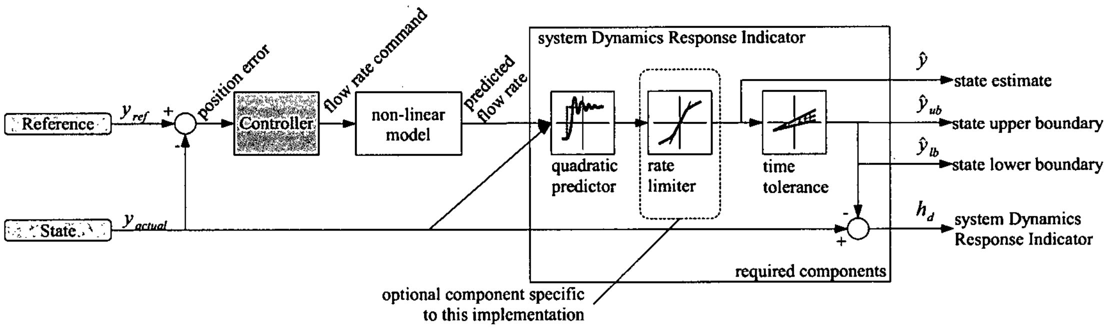


Figure 4.7: Dynamics Response Indicator

In Figure 4.7, the output from the non-linear model of Figure 4.6 is shown entering the Dynamics Response Indicator. The linear part of this model is the quadratic predictor, which produces three outputs: a high threshold, low threshold, and a prediction of the system state. Each of these signals is passed through a rate limiter followed by a time-tolerance. The prediction of system state is the state estimate output of the health predictor. This, combined with the high and low thresholds make up the analytical redundancy that would be used to reduce degradation given a sensor failure. The DRI is calculated from the difference between the actual system state and the high or low boundary of system state, whichever yields the smaller magnitude.

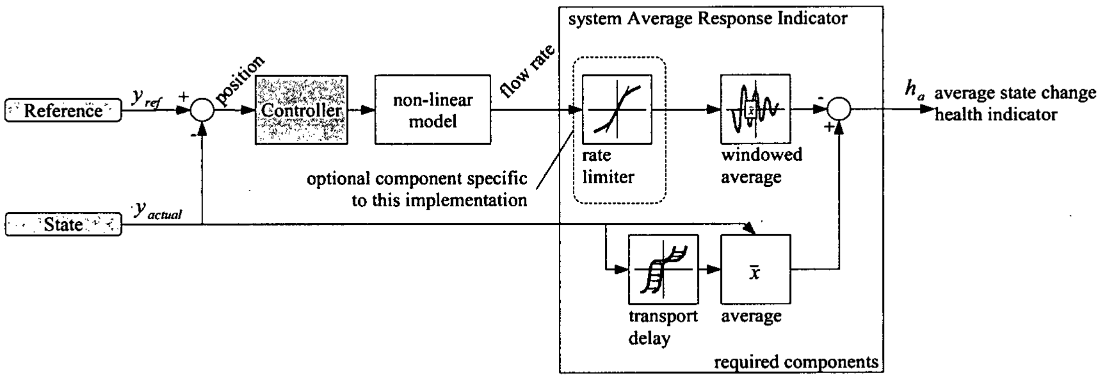


Figure 4.8: Average Response Indicator

In Figure 4.8, the output from the non-linear model of Figure 4.6 is shown entering the Average Response Indicator. The value passed into the health indicator is an estimate of the flow rate, but it does not take in to account the delay that is caused by switching the valve direction. The rate-limiter object is used account for the switching delay, and the output from it represents the cylinder velocity.

The average state change indicator is the difference between the modelled average cylinder velocity over a period of time, and the average measured velocity at the start and end of the same time interval.

4.3.2 Hydraulic System Identification

The behaviour of the hydraulic cylinder in response to the controller input was identified using Plackett's algorithm for least squares system identification, but only after the non-linear elements were cancelled using the non-linear model described in section 4.3.1. To verify the findings of this system identification, the data was checked again in Matlab, using the system identification toolbox.

The system undamped natural frequency was identified as 70 rad/s (average of 40 tests), and the damping ratio was identified as 0.7 (average of the same 40 tests).

4.3.3 State Predictor

The State Predictor sensor provides an analytically redundant system sensor to enhance detection of sensor faults and protect against continuous system degradation. It provides a best prediction of the system state using the same model and object as the DRI, and uses the DRI's upper and lower health thresholds as limits on the accuracy of the sensor, which makes it more sensitive when the system is in a static condition, and less sensitive during dynamic conditions.

Section 3.2.4 discusses the redundancy management architecture and the Local Error Detection class. The Local Error Detection object uses the information about the accuracy and range from each sensor that is assigned to a given Quantity. The sensor information is used to perform an inexact voting scheme to determine if any sensor has failed, and then a weighted average to calculate the Quantity state value. The output from the analytical sensor can be used by the Local Error Detection object in the same way. If the history of the Dynamics Response Indicator shows that the system is in good health, then this sensor maintains the voting functionality required by the Local Error Detection routines. However, once the DRI is being used as a sensor, the value of the health monitor for that component becomes questionable, as each state used by the model is dependent on a previous model estimate.

To validate this sensor, its output is compared to the acquired sensor values and accuracy limits for the Local Error Detection object. If the Local Error Detection routines do not detect it to be faulty for the normal case then it provides an adequate state prediction during normal operations. The sensor must then be validated for sensitivity in each of the simulated fault operating conditions.

4.4 Validation Parameters

4.4.1 Repeatability

To validate the health indicator for repeatability, a number of tests were run under various operating conditions, input frequencies and input magnitudes. Each of the indicators is inspected graphically to determine whether it is performing in a repeatable and expected manner. The magnitude of the error signal recorded for the duration of the test is examined to verify that indicated errors can be confirmed. The only errors introduced in this test are power failure, low flow, and fast input signal.

4.4.2 Sensitivity

To validate the health indicator for sensitivity, the output from the health indicator is checked against varying operating conditions. The tests varied flow rate and pressure of the hydraulic system, allowing for errors to be simulated for detection.

4.5 *Test Setup*

4.5.1 Health Monitoring Configuration

The embedded computing system was run with a loop closing and sampling interval of 4 ms. The health monitoring model was configured with a prediction horizon of 4 samples, for a prediction time of 16 ms. Given the upper bound of frequency at 14.3 Hz (70 rad/s), this represents approximately 82 degrees of motion. The parameters used by the dynamic model for the tests are shown in Table 4.1. A complete list of model parameters is given in Table C.1.

Table 4.1: Health Monitor Parameters

Parameter	Value	Unit
Quadratic Predictor		
k (k)	1	
wn (ω_n)	70	rad/s
wn_l (ω_{lb})	40	rad/s
wn_h (ω_{ub})	90	rad/s
zd (ζ)	0.5	
zd_l (ζ_{lb})	0.4	
zd_h (ζ_{ub})	0.8	

4.5.2 Test Input

The input is a computer-generated signal that represents a position command as would be received from the helm, where a given input position is proportional to an output position in the cylinder. The input signal is already scaled to represent a command of the cylinder position, to set the system gain to 1.

To identify the affects of all operating conditions, (particularly transitions in the valve state) the input signal used is a triangular wave, of constant amplitude, but at varying periods. The selected input periods are 1500 ms, 1000 ms, 500 ms, and 333 ms, corresponding to input frequencies of 0.67 Hz, 1.0 Hz, 2.0 Hz, and 3.0 Hz.

4.5.3 Operating Conditions

The system is tested under three different operating conditions. The first condition is normal operation, and is used as the benchmark for all tests. Under normal operating conditions, the motor is fully powered, and the manual valve shown in Figure 4.3 is fully closed. The steering cylinder has no load to move. The current model and control of the pump output is strictly on-off, and for this thesis, it was assumed that the pump's output flow rate was at the saturation point during testing.

In the second operating condition, the system undergoes intermittent power failure. The intermittent power failures are achieved by switching off the motor power at the solid state relay indicated in Figure 4.5. The relay is switched off at ¼ of the way through the second triangular wave, and restored at ¼ of the way through the third wave. This process is repeated for each of the four wave periods, creating intermittent power failures of varying times, but proportional by time within each input frequency.

The third operating condition has the system run in a low-flow state. The manual valve shown in Figure 4.3 is opened part way, which re-routes the flow away from the control valve, and therefore decreasing flow rate at the cylinder. The manual valve is not adjusted during the test, so the flow loss is constant.

Figure 4.9 to Figure 4.23 show data collected from three tests, one from each of three operating conditions. The first figure in each set is under normal operating conditions, followed by graphs of the system experiencing intermittent power failure, and finally experiencing a low-flow operating condition.

4.6 Instantaneous Response Indicator – System Position

The Instantaneous Response Indicator is shown in Figure 4.9 to Figure 4.11. The ability of the system to track errors in the normal operating conditions and also in the error conditions is shown. This error signal clearly shows that the system plant has a poor ability to track a given input signal.

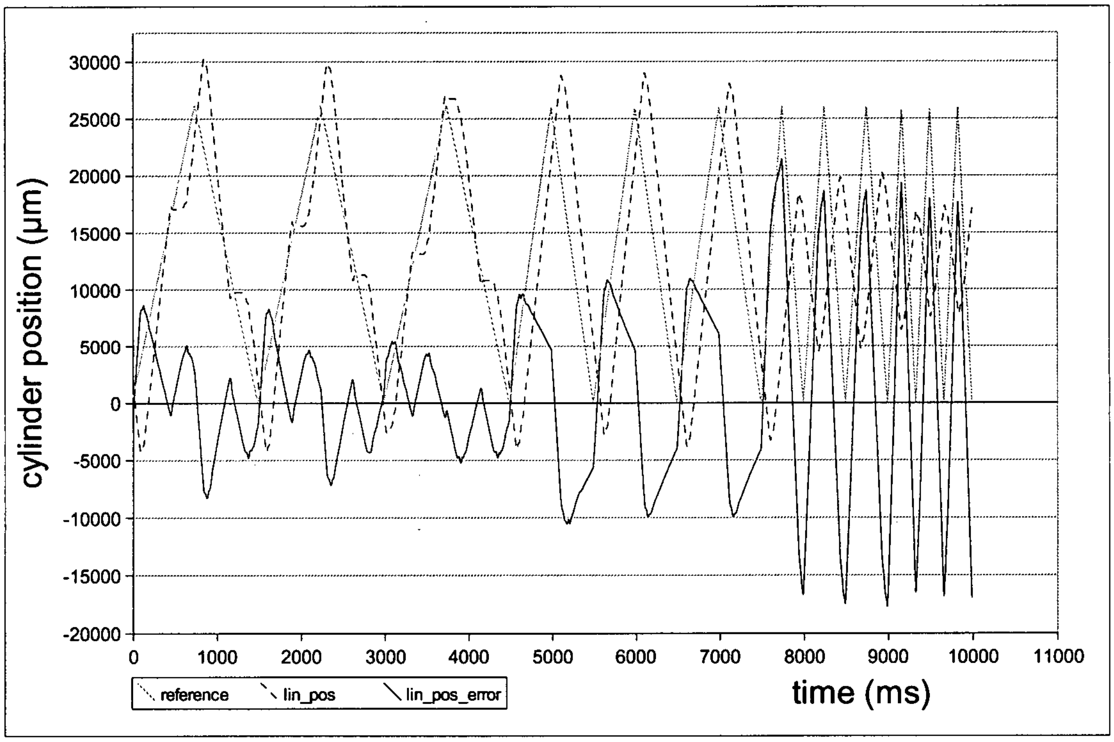


Figure 4.9: Instantaneous Response Indicator: Normal Operation

In Figure 4.9, the low frequency triangular wave input test shows the system tracking to within a maximum error of 6 mm. Over the input range of the signal, 26.250 mm, this represents a margin at 23% of the motion. On the assumption that this is an acceptable input, then the threshold for poor health detection would be around 6mm. When the test input period decreases to 1.0 s, one can clearly see an increased error margin; with an input period of 0.5 s or 0.333 s, the system response moves out of phase with the input and large errors occur through all tests. The errors indicated with input period of 1.0 seconds demonstrate that the system cannot respond to the magnitude of the motion command, as the velocity command is now faster than the saturation limits provided by the pump.

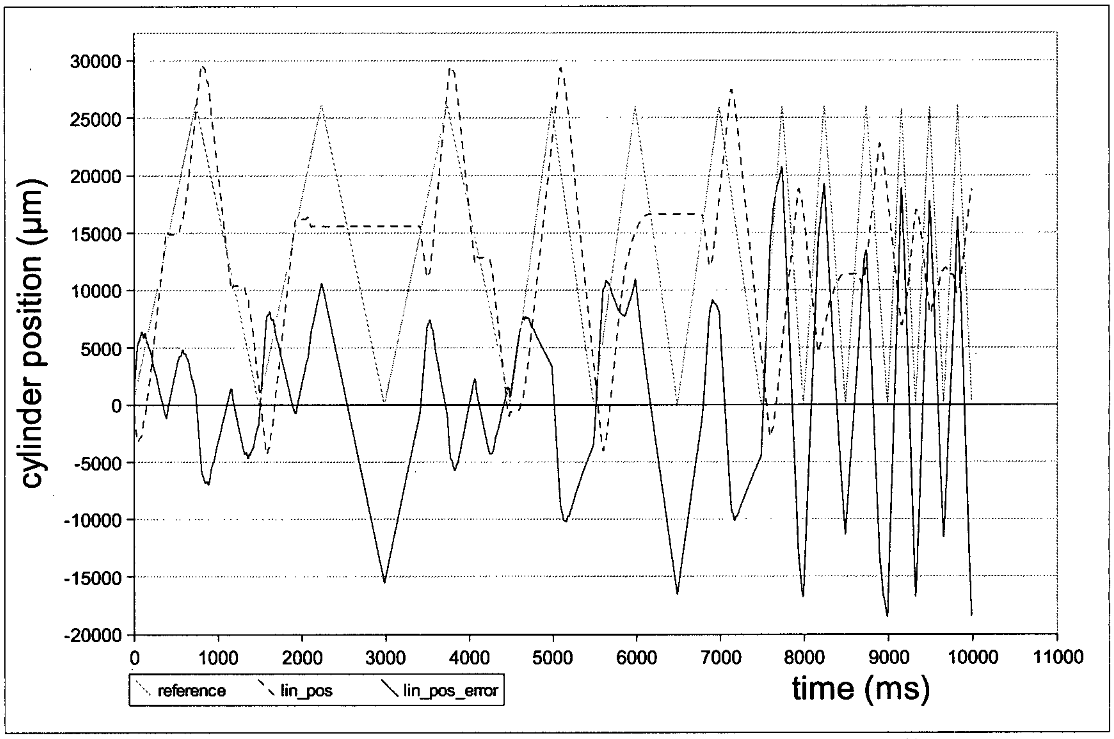


Figure 4.10: Instantaneous Response Indicator: Intermittent Power Failure

Figure 4.10 shows the IRI during a test with the injection of an intermittent power failure fault. The indicator shows similar output compared to the no-fault test, until a power failure is injected. When the power failures are injected, the indicator's magnitude increases to as much as 55% of full scale, clearly indicating poor system health.

Figure 4.11 shows the IRI during a test with the low-flow error condition; throughout this test, the magnitude of the indicator remains large (greater than 25% of full scale), and has less variation than during the no-fault test.

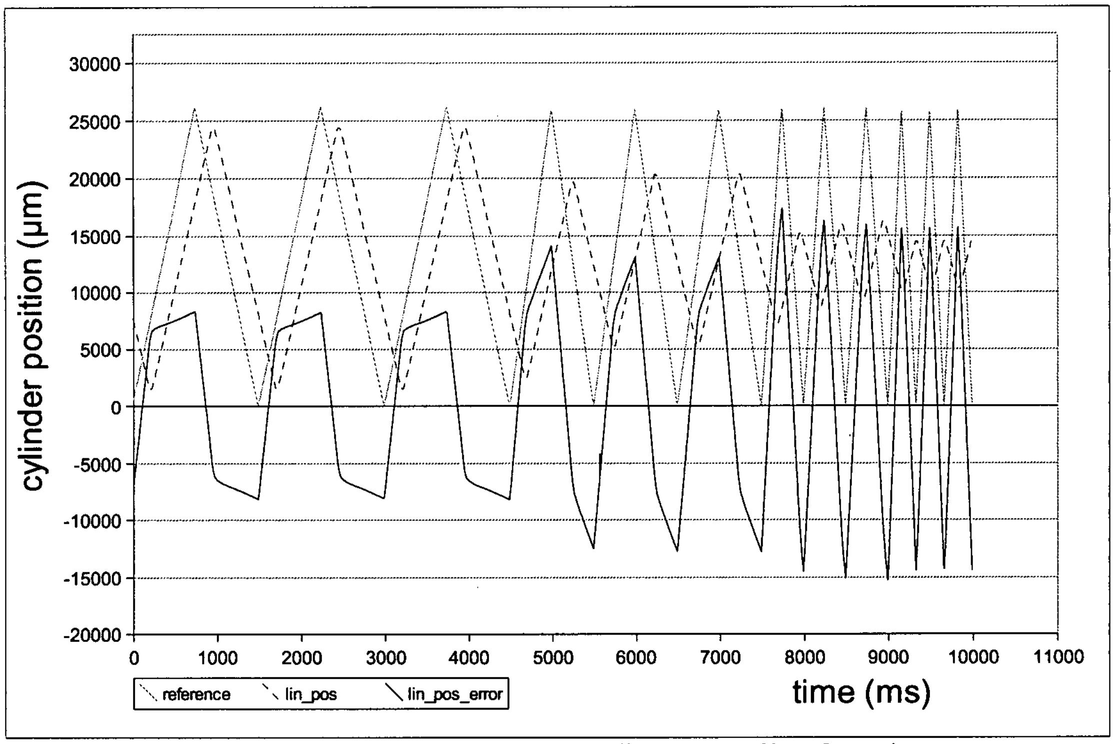


Figure 4.11: Instantaneous Response Indicator: Low Flow Operation

4.7 Average Response Indicator – System Average Velocity

The results for the Average Response Indicator are shown in Figure 4.12 through Figure 4.14. Each figure shows three data series. The first series is the measured velocity of the hydraulic cylinder. The error signal is the difference between the actual velocity and the expected velocity, which is developed in Chapter 3. As shown in Table 4.1, the system velocity is averaged over 0.120 seconds. At the leading edge of each step one can see a significant error margin appear. During this dynamic range, where expected frequencies are in the range of 10 Hz to 20 Hz, this is expected, but when the actual velocity begins to settle, one can easily detect errors between the modelled and actual velocities. One can also note that the velocity behaviour is significantly different when alternating the direction of flow compared to alternating the flow between 0 and saturation. One can observe in these figures that the model is able to track changes in the modelled input response for the normal case, but also detect errors that affect, in this case, flow rate.

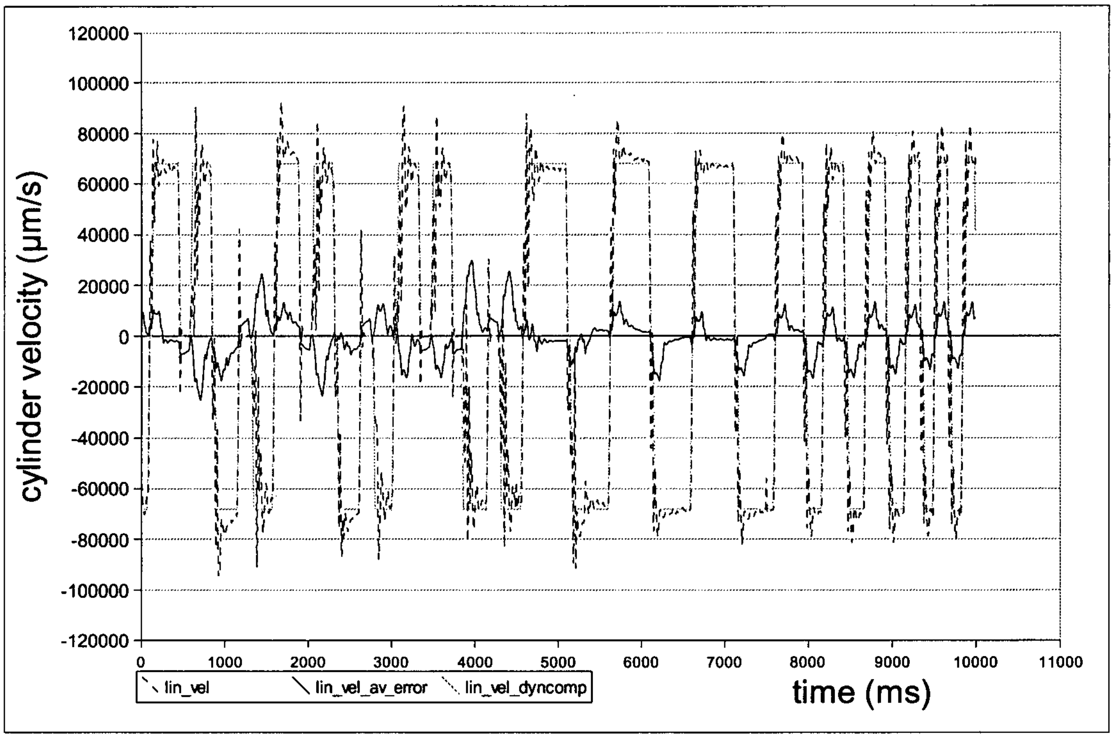


Figure 4.12: Average Response Indicator: Normal Operation

Figure 4.13 shows the ARI during a test with the injection of an intermittent power failure fault. In this test, the disparity between the expected behaviour and the actual behaviour is dramatic, and the signal from the health indicator clearly shows poor health.

Figure 4.14 shows the ARI during a test with the low-flow error condition. During this test, the magnitude of the signal from the health indicator is consistently large (about 50% of full scale), and clearly indicates poor system health.

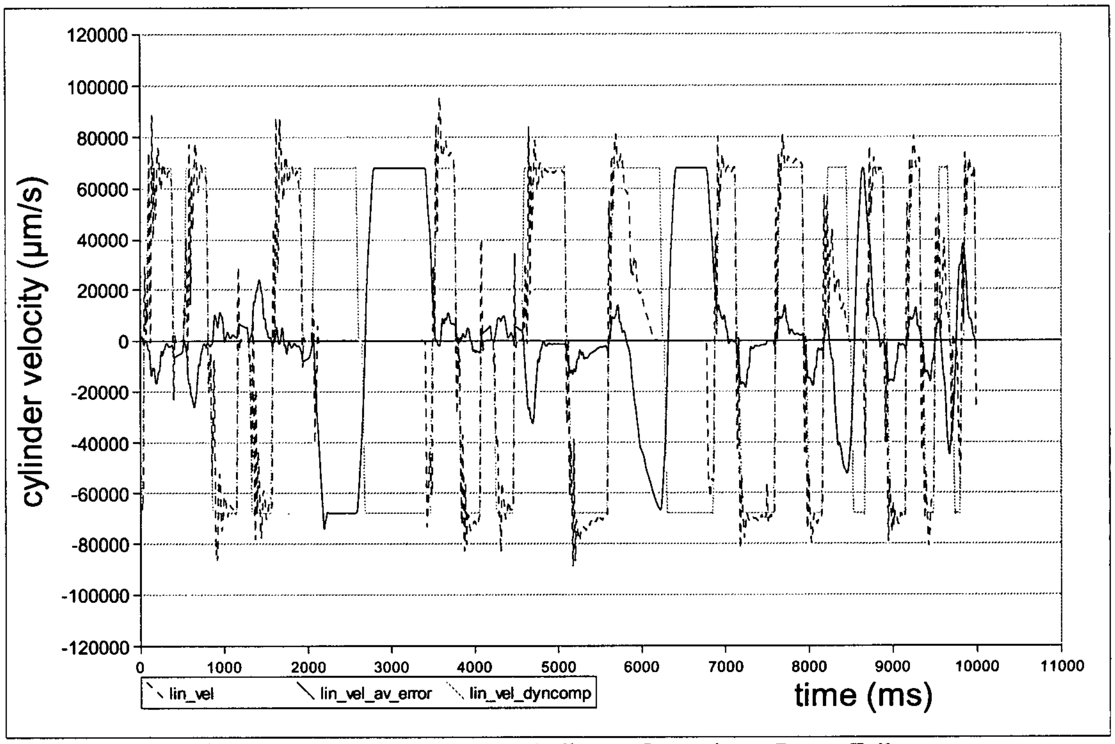


Figure 4.13: Average Response Indicator: Intermittent Power Failure

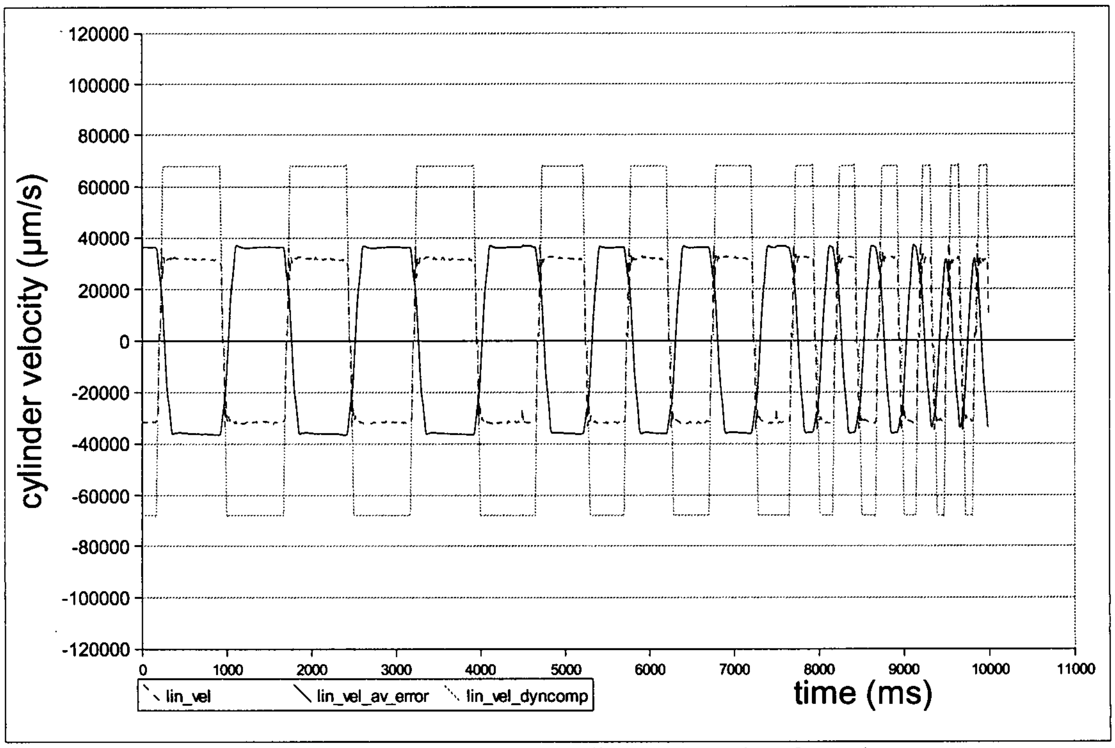


Figure 4.14: Average Response Indicator: Low Flow Operation

4.8 Dynamic Response Indicator – System Velocity

One can observe in Figure 4.15 to Figure 4.17 that the Dynamic Response Indicator shows little response to normal operating conditions, thus indicating that the model can accurately predict boundaries on system dynamic response. In Figure 4.18 to Figure 4.20, one can see how the Dynamic Response Indicator immediately responds to changes in dynamics through the intermittent power failure test, and Figure 4.21 to Figure 4.23 show results for the low flow operating condition.

4.8.1 Normal Operations

Figure 4.15 shows 10 seconds of test data under normal operating conditions. Figure 4.16 shows the same data set during the interval 2.0 seconds to 3.5 seconds[4]. During this period, the position input signal is a triangular wave pattern with a period of 1.5 seconds; the exact input data can be seen in Figure 4.9. These figures show the upper and lower boundaries output by the Dynamic Response Indicator, the actual system velocity, and any error encountered. The figures show the extent of the non-linearities encountered, the irregular settling velocity, peak velocities, and dynamic responses of the cylinder. In particular, one can note the settling characteristics of the system at times 2300 ms and 2800 ms. The responses at these times exemplify the large variation in system dynamics when responding to similar commands from different starting states. Also, one can notice the lack of symmetry with each of the responses, the different peak values, and different settling values.

4 The period from 2.0 seconds to 3.5 seconds has a relatively slow input signal (longer period of the square wave). The same period of time is shown in each of the subsequent tests, when faults are injected into the system.

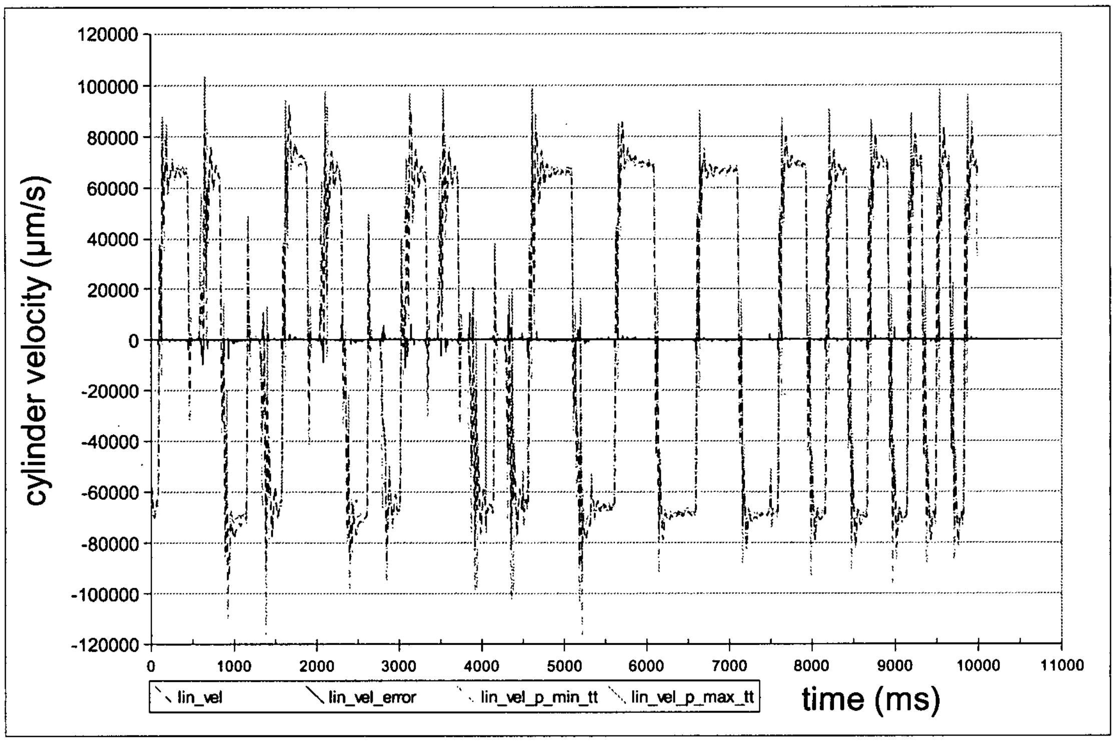


Figure 4.15: Dynamic Response Indicator: Normal Operation

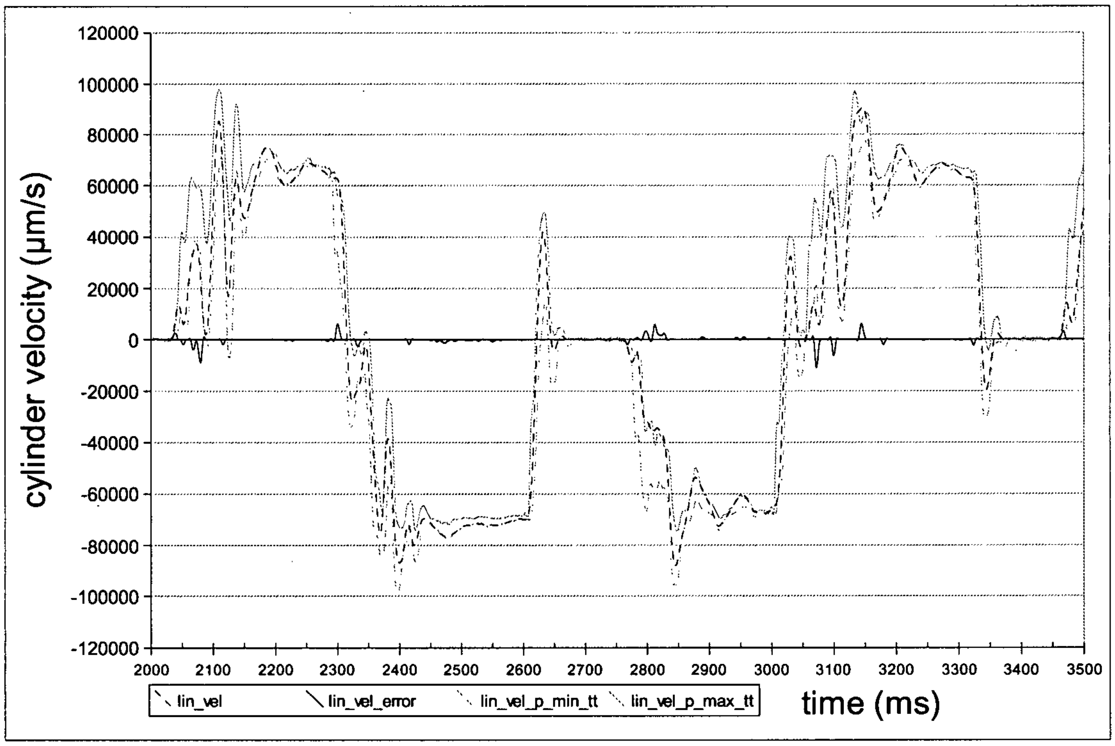


Figure 4.16: Dynamic Response Indicator: Normal Operation Slow Input

Finally, in Figure 4.17, which is of the same test at time interval 8.0 seconds to 9.0 seconds[5], one observes that the system responses are fairly similar at each step, and while there is a recurring error detected at each leading edge, it lasts for only one cycle. The system responses are nearly symmetric across the time axis, and while there are slight differences in the settling velocities, the Dynamic Response Indicator handles these differences without issue.

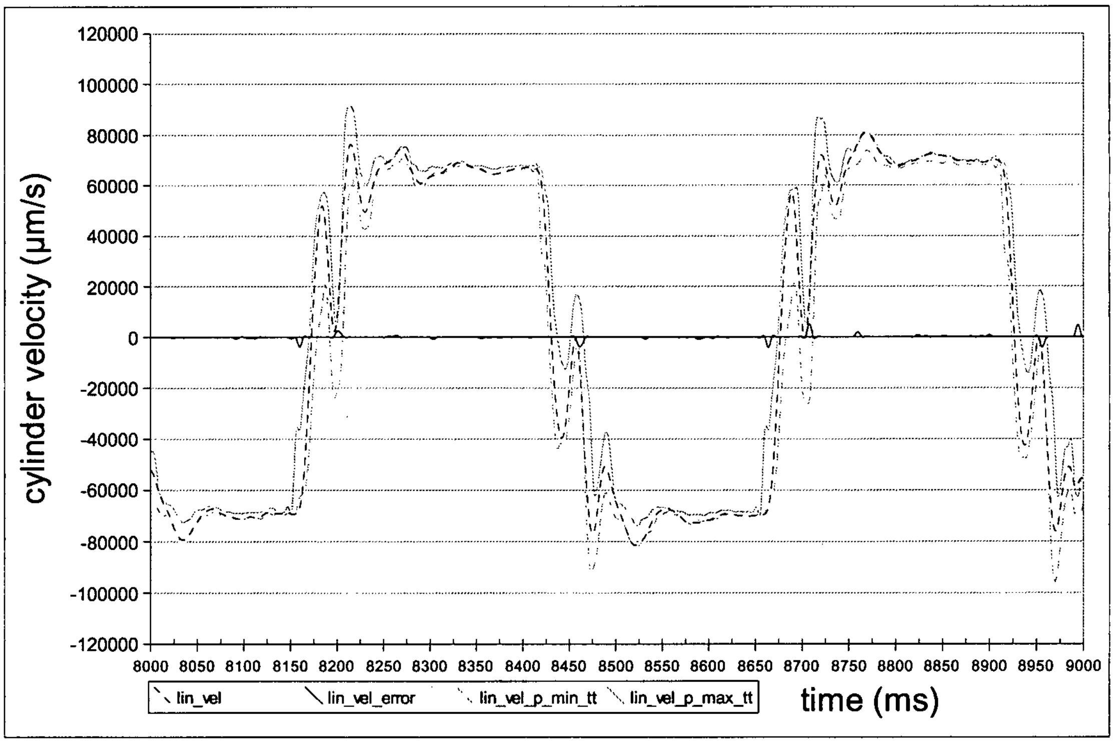


Figure 4.17: Dynamic Response Indicator: Normal Operation Fast Input

In these graphs, one can now clearly see the four data sets. The actual system state generally falls within the predicted high and low boundaries of the Dynamic Response Indicator. The error signal usually occupies the time axis except for a few short deviations.

5 The period from 8.0 seconds to 9.0 seconds has a relatively fast input signal (shorter period of the square wave). The same period of time is shown in each of the subsequent tests, when faults are injected into the system.

4.8.2 Intermittent Power Failure Operating Condition

Figure 4.18 shows 10 seconds of test data when the hydraulic pump and motor system experiences intermittent power failures. The significant behaviour change that results from the un-powered condition is clearly visible in the figure, and is apparent at times 1900 ms to 3400 ms, again at 6000 ms to 7000 ms, 8200 ms to 8700 ms, and at 9400 ms to 9700 ms. Figure 4.19 shows the same data set during the interval 2.0 seconds to 3.5 seconds. During this phase, the position input signal is a triangular wave pattern with a period of 1.5 seconds; the exact input data can be seen in Figure 4.10. These figures show the upper and lower boundaries output by the Dynamic Response Indicator, the actual system velocity, and any error encountered.

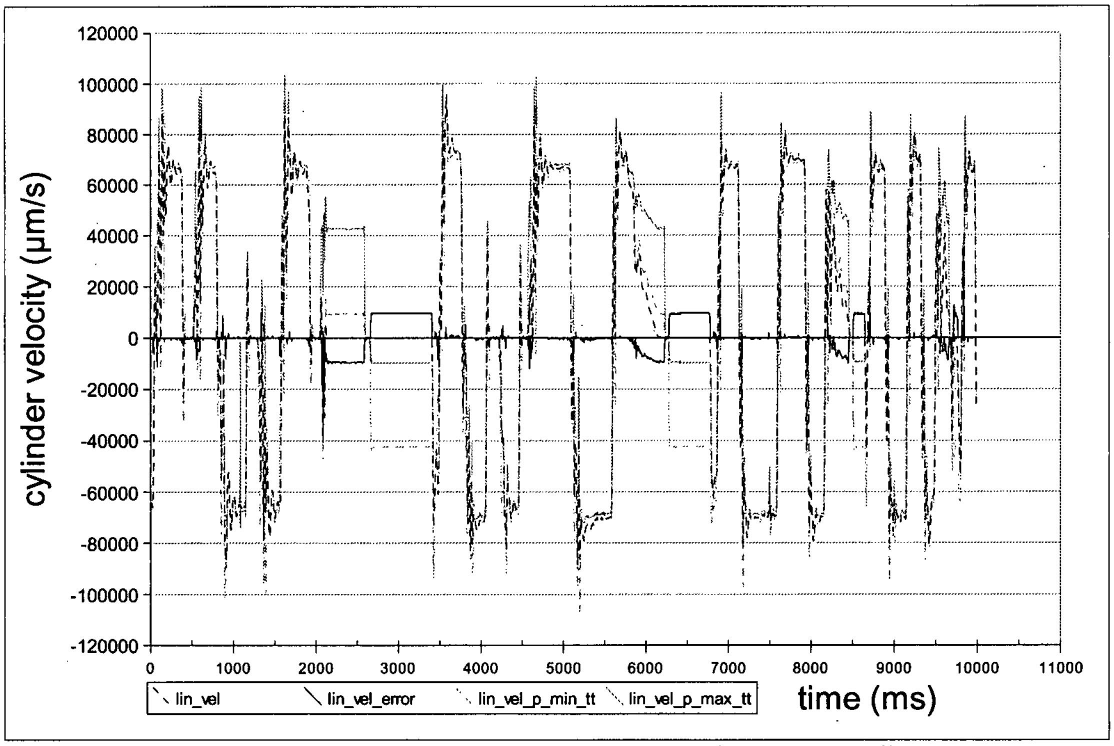


Figure 4.18: Dynamic Response Indicator: Intermittent Power Failure

During each period immediately following the leading edge of a step input, the system quickly detects large deviations from the dynamic model, this is shown in Figure

4.19 as two negative spikes. The system also shows a significant settling velocity error. During the transition phase between a positive and negative step, one can see that the system stops detecting errors; this is expected behaviour for this indicator, as the refresh of starting data briefly places the system velocity at 0 with a command velocity of 0. Figure 4.20 shows the same test at the time interval 8.0 seconds to 9.0 seconds, and captures the power failure when the system is still in the dynamics phase. The error indicator immediately begins to grow as the system velocity drops to 0. Like the slow input case, there is a brief transition period as the valve switches flow direction, before reaching a new settling error.

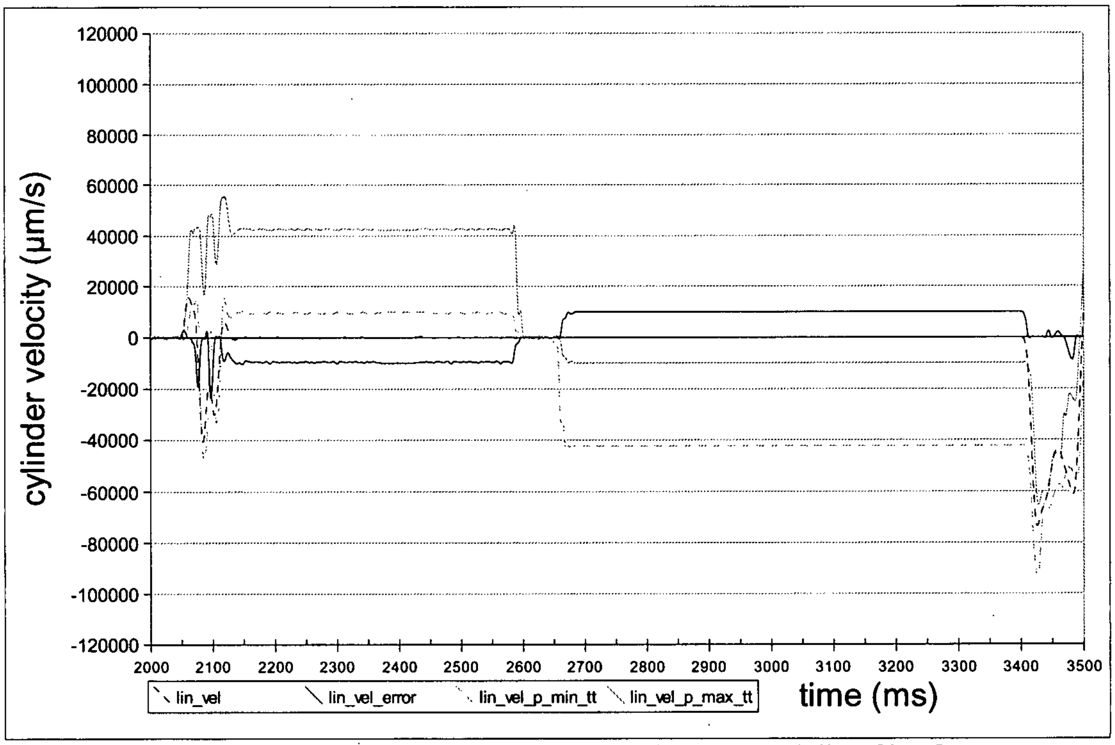


Figure 4.19: Dynamic Response Indicator: Intermittent Power Failure Slow Input

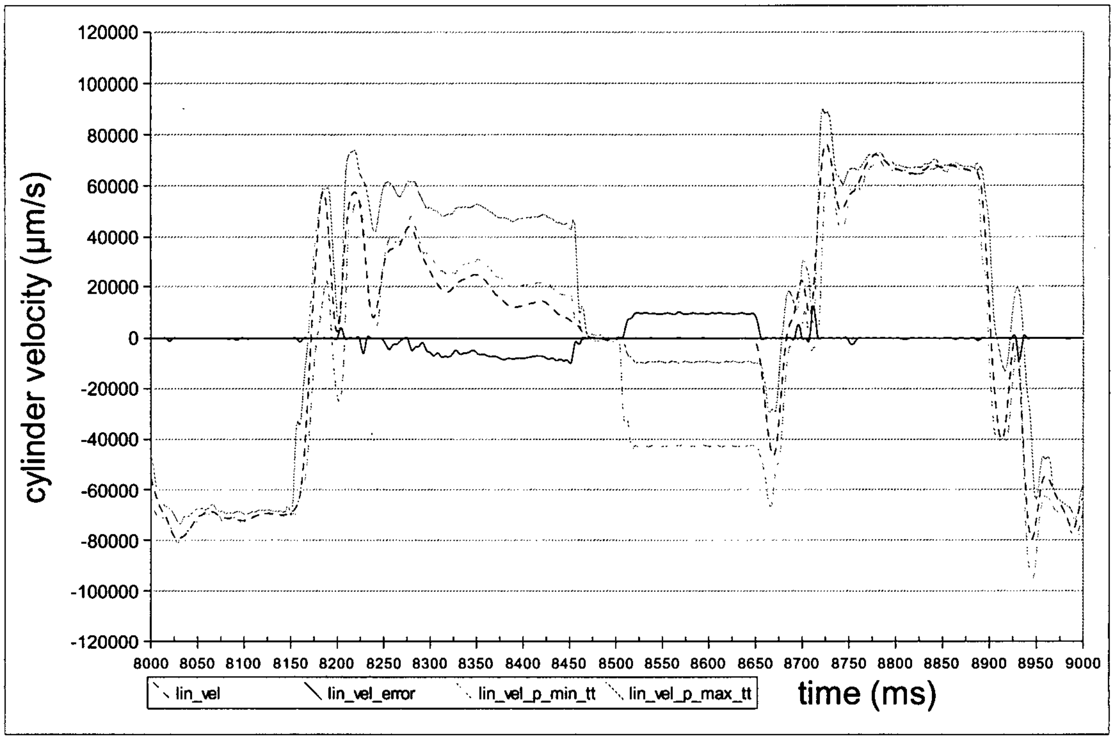


Figure 4.20: Dynamic Response Indicator: Intermittent Power Failure Fast Input

4.8.3 Low Flow Operating Conditions

Figure 4.21 shows 10 seconds of test data when the hydraulic steering system experiences a low flow condition. The low flow condition is comparable to a significant system leak, power supply reduction to the motor, or a pump problem. The low flow condition is created by partially opening the manual relief valve, which is located before the 3-position control valve and feeds back to the reservoir. The low flow condition is easily detected by the indicator, and can be clearly seen when inspecting the figures visually; actual system speed is always less than the predicted speeds, and generates a significant steady state error.

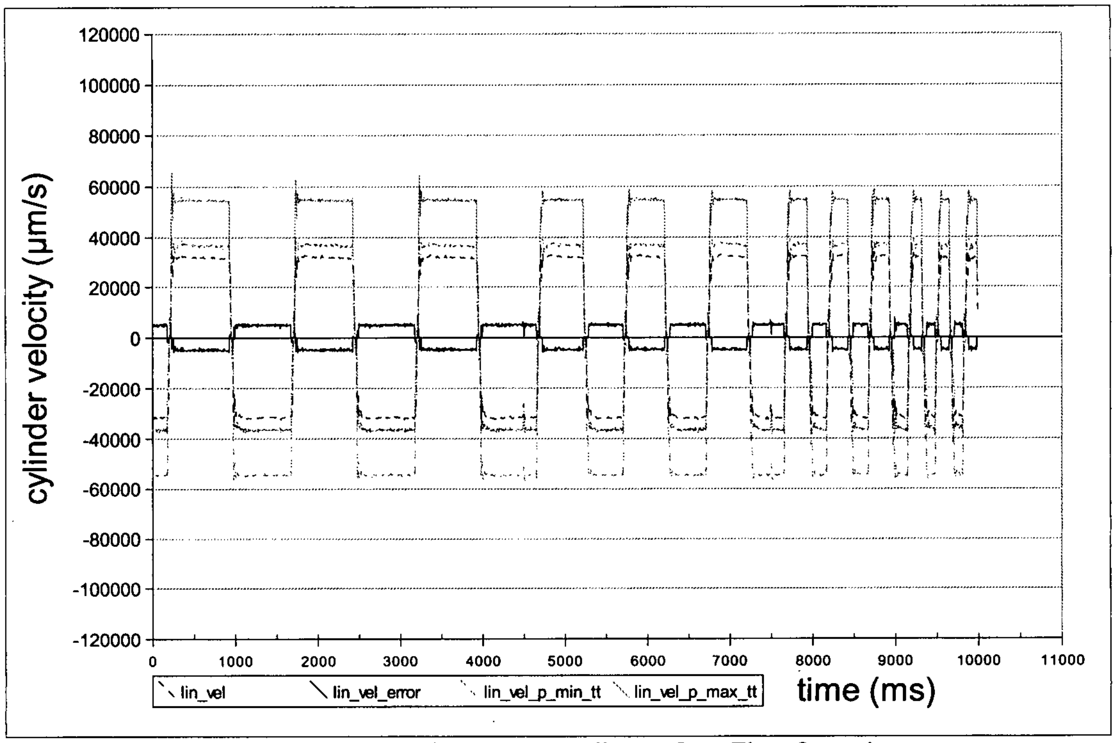


Figure 4.21: Dynamic Response Indicator: Low Flow Operation

Figure 4.22 shows the same data set during the interval 2.0 seconds to 3.5 seconds. During this period, the position input signal is a triangular wave pattern with a period of 1.5 seconds; the exact input data can be seen in Figure 4.11. These figures show the upper and lower boundaries output by the Dynamic Response Indicator, the actual system velocity, and any error encountered.

During each period immediately following the leading edge of a step input, the system's dynamic response is within the tolerances of expected behaviour. This occurs because the dynamics (undamped natural frequency, damping ratio) are relatively similar, but the gain is lower. However, the Dynamic Response Indicators shows a significant settling error. Figure 4.23 shows the same test at the time interval 8.0 seconds to 9.0 seconds, and clearly shows that system dynamics still behave as predicted. As it did in the power-off case, the system reaches a settling velocity error in this test too.

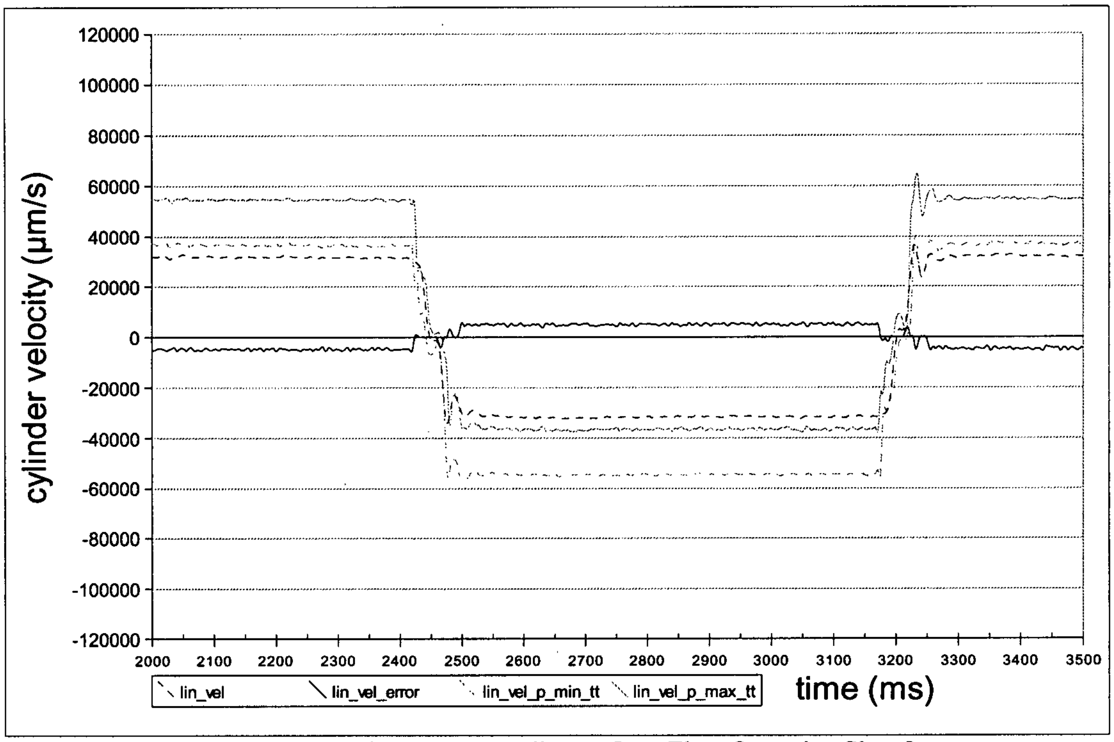


Figure 4.22: Dynamic Response Indicator: Low Flow Operation Slow Input

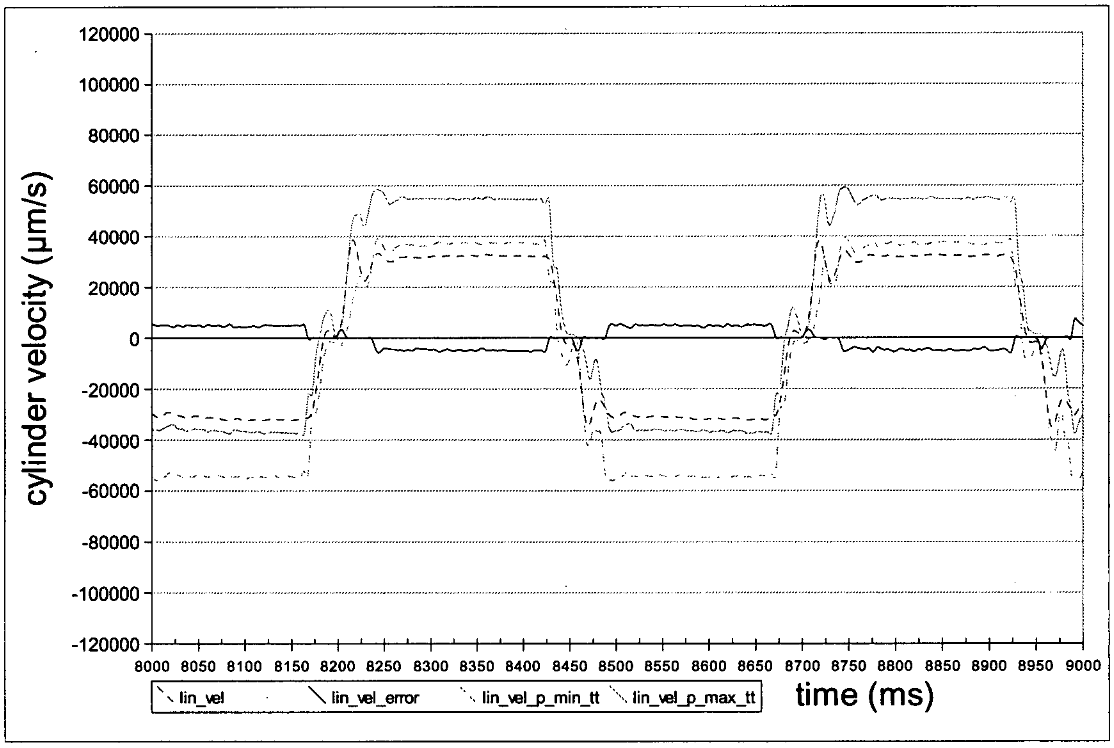


Figure 4.23: Dynamic Response Indicator: Low Flow Operation Fast Input

4.9 ***Results Achieved and Limitations***

In the hydraulic steer-by-wire example, the health monitoring system has been applied to a single system and sensor. The health monitoring system evaluates the performance of the position and velocity of the hydraulic cylinder with respect to the helm input, and the actuator inputs at the pump and valve. The sensed parameter that is augmented with analytical redundancy is the linear velocity of the cylinder. The hydraulic steering system simulator was designed, assembled and instrumented.

The health monitoring indicators detect healthy operation when the system is operating normally, with typical inputs. Typical input to the system is represented by the 0.67 Hz and 1.0 Hz triangular wave inputs.

The indicators have been tested against two error conditions. The first error condition is the power-off case. In this case, the Instantaneous Response Indicator shows a rapid increase in error, but also shows a decrease in error as the reference signal passes through the stalled state of the system. The Average Response Indicator shows a large error at each of the power-off cases. The Dynamics Response Indicator however, shows that the system dynamics stay within operating requirements at the beginning of a power failure, but it also shows a steady state error. It also clearly shows that the dynamics do not perform within specification after a short period of the power-off.

The second injected error condition is the low-flow case. Here, the Instantaneous Response Indicator shows a larger response (which demonstrates poor health) than in the normal operations. The Average Response Indicator shows a large errors response throughout all of the periods of steady-state operation. The Dynamic Response Indicator shows that the steering system continues to behave within the specifications in the interval immediately following a change at the valve, where the system dynamics are more pronounced. It also shows that the steering system is unhealthy during the steady-state period.

Finally, the Dynamic Response Indicator results show that this indicator's performance also makes it adequate for use as a redundant sensor tool for the system velocity, should a redundant velocity sensor fail. The output from the DRI is based on previously measured values from the redundant sensors, when they are considered healthy. However, the accuracy of the DRI as a sensor is less than the physical sensors, its value would not be trusted alone. Instead, it would be included in the voting scheme to arbitrate between two sensors that do not agree after the third has failed.

One limitation of this work is its need for further testing of the sensitivity of the health indicators. The tests show that the data output from the health indicators is a good predictor of system state, and that the health monitoring parameters can be selected to detect when the system behaves as expected. The tests show that the system can detect low power scenarios, low flow scenarios, and also shows errors when the system responds to high frequency input.

The health monitoring system is designed to use data collected from multiple ECUs, where each of the system states are measured and shared across a network. The indicators selected are designed that they can also easily be shared across a network. The distributed functionality of the system states was developed in the preceding work by Bouvier. The communication of the health monitoring states is also untested in the network.

Other system states could be monitored for errors, particularly within the hydraulic system. Ideally, the pressure transducers could be used to measure the system pressure throughout the system, and monitor pressure change across the valve, and then this could be used within the system model to predict the acceleration at the cylinder. The pressure transducers would decrease the granularity of errors that are detectable by the health monitor. The existing system is limited in granularity by the position of sensors used; only the entire hydraulic system from pump to cylinder is included in the model. Instrumentation within the hydraulic system would allow for errors of finer granularity to be detected.

No diagnostic module has yet been developed to assess when the health indicators' output detects failure, and this needs to be developed for the system to be a practical tool as an actuator performance measure.

4.10 Summary

In this chapter, the concepts detailed in Chapter 2 and Chapter 3 are shown in use in an experimental system that was designed and built as part of this thesis. The experimental apparatus is described in detail.

The concepts of condition monitoring, parameter identification, and state estimation are put to use within the health monitoring scheme. The system models used and the parameters used within the software written are shown. This chapter also describes the identification of the hydraulic system.

Experiments performed to verify the presented theory are described. Inputs to the experiment and the health monitor are shown, and graphs of the experimental results are shown. The health indicators are tested in normal operation, and then with injected errors: intermittent power failures and low pump flow.

Finally, the performance of the indicators of the experimental system is assessed, and some of its limitations are indicated. In particular it shows that the combination of the three health indicators can indicate error conditions when appropriate, and that the model used by the Dynamic Response Indicator can be used as a redundant analytical sensor arbitrator, if a redundant physical sensor fails.

Chapter 5: Conclusions and Recommendations for Future Work

5.1 Conclusions

The use of embedded systems in consumer and industrial applications is expected to increase in the future. The performance, flexibility, efficiency and feature benefits associated with these integrated systems are too desirable to ignore. However, one must acknowledge the potential for risk involved in these systems, and then realize that to gain public acceptance and guarantee the stringent safety requirements, it is necessary to build an architecture that provides a high level of fault-tolerance. It is also evident that these systems must have an awareness of the health of each of the components. Health monitoring adds a layer of safety, and it also adds the potential to increase uptime and protect the monitored system against degradation.

Other researchers have approached this task with high levels of redundancy, specific sensors to detect health, and computationally expensive techniques. This thesis shows that the uses of analytical techniques in an appropriate framework can adequately detect the health of actuators, model their performance, and predict their future states, without the burden of adding specific sensors and without adding high computational loads. The architecture presented in this thesis combines local error detection, distributed error detection, health indicators and system modelling.

Health monitoring systems not only provide indicators of health, but also integrate adaptive thresholds, and provide analytical redundancy that protects the system against continuous degradation when sensor failure occurs. The health indicators can compose a system-wide health vector that indicates general system health or a specific component's health.

The health monitoring framework uses a novel approach of using continuously updated non-linear system models, paired with reduced order system modelling and performance criteria to provide composite health indicators with adaptive thresholds and analytical sensor redundancy.

Using this approach it is possible to detect actuator, plant, and active component degradation and errors, and function when a sensor has failed.

A hydraulic marine steer-by-wire system is a typical example of an integrated vehicle by-wire system, and can therefore be used to illustrate the concepts described by this thesis. An experimental prototype of such a system was built by the author as part of this thesis. The health monitoring architecture described was implemented. Testing has shown that the system is capable of representing normal behaviour, and also of detecting typical errors that occur in such a system.

5.2 Recommendations for Future Work

This work produces a series of health indicators, and a framework for organizing them. It also provides analytical sensor redundancy. The diagnostic functions were not developed as part of this thesis, and would be required to adequately compensate for actuator faults, and recommend maintenance intervention. The actuation management class supports multiple controllers for a single plant, but this functionality has not been implemented in practise. Robust algorithms to allow for shared actuator loading should be investigated and applied to this project.

The existing fault-tolerant architecture does not include analytical redundancy with adaptive thresholds within the local error detection and quantity error detection classes. To make full use of the health monitoring layer, these classes should be revised.

Finally, the health monitoring layer uses shared data from each of the ECUs. As a required function however, a health monitoring distributed error detection class should be developed to ensure that the critical functionality and permissions granted to the health monitoring system, subsequent diagnostic and actuation management be themselves, fault-tolerant.

Bibliography

1. Frank, P. M., Ding, X., 1997, "Survey of robust residual generation and evaluation methods in observer-based fault detection systems," Journal of Process Control, Vol. 7.6, pp. 403-424.

2. Hiller, M., 1999, Thesis: "Using Software to Handle Data Errors in Embedded Control Systems," Chalmers University of Technology.

3. Laprie, J. C., "Dependability - Its Attributes, Impairments and Means," Predictably Dependable Computing Systems, pp. 3-24, Springer-Verlag, Berlin; Heidelberg; New York, 1995.

4. Koopman, P., 2003, "Elements of the Self-Healing System Problem Space," ICSE WADS03, pp. .

5. Askerdal, O., Gafvert, M., Hiller, M., Suri, N., 2003, "Analyzing the Impact of Data Errors in Safety-Critical Control Systems," IEICE Transactions on Information and Systems, Vol. E86-D.12, pp. 2623-2633.

6. Noura, H., Sauter, D., Hamelin, F., Theilliol, D., 2000, "Fault Tolerant Control in Dynamic Systems: Application to a Winding Machine," IEEE Control System Magazine, pp. 33-49.

7. Mutuel L. H., Speyer J. L., 2000, "Fault Tolerant Estimation," Proceedings of the American Control Conference, pp. 3718-3722.

8. Parhami, B., 1994, "Voting Algorithms," IEEE Transactions on Reliability, Vol. 43.4, pp. 617-629.

9. Blough, D., Sullivan, G., 1990, "A Comparison of Voting Strategies for Fault-Tolerant Distributed Systems," IEEE Proceedings - Symposium on Reliability in Distributed Software and Database Systems, pp. 136-145.

10. Shelton, C., Koopman, P. & Nace, W., 2003, "A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems," WORDS 2003, pp. .

11. Bouvier, M., 2002, Thesis: "Definition of a Cost-Effecive, Fault-Tolerant Control Architecture: application to the Design of a Steer-by-Wire System," University of British Columbia.

12. Zhou, D., Frank, P., 1998, "Fault Diagnostics and Fault Tolerant Control," IEEE Transactions on Aerospace and Electronic Syst, Vol. 34.2, pp. 420-427.

13. Caliskan F., Hajiyev, CH. M., 2000, "EKF Based Surface Fault Detection and Reconfiguration in Aircraft Control Systems," Proceedings of the American Control Conference, Vol. ., pp. 1220-1224.

14. X-By-Wire team, 1998, "X-By-Wire, Safety Related Fault Tolerant Systems in Vehicles, Final Report," http://www.vamars.tuwien.ac.at/projects/xbywire/do accessed: 05/09/2003.

15. Das, 0., Woodside, C., 2002, "Modeling the Coverage and Effectiveness of Fault-Management Architectures in Layered Distributed Systems," Proceedings of the International Conference on Dep, Vol. ., pp. .

16. Fitch, J. C., 1986, "Systems and Methods for Real-Time Condition Monitoring of Mechanical Machinery," Society of Automotive Engineers, pp. .

17. Goode, K. B., Moore, J., Roylance, B. J., 2000, "Plant machinery working life prediction method utilizing reliability and condition-monitoring data," Proc Instn Mech Engrs, Vol. 214.E, pp. 109-122.

18. Yunbo, H., Chua, P. S. K., Lim, G. H., 2002, "Online Monitoring the Condition of Loaded Water Hydraulic Actuators," SAE Technical Paper Series, pp. .

19. Rioul, O., Vetterli, M., 1991, "Wavelets and Signal Processing," IEEE Signal Processing Magazine, pp. 14-38.

20. Wang, W. J., McFadden, P. D., 1996, "Application of Wavelets to Gearbox Vibration Signals for Fault Detection," Journal of Sound and Vibration, Vol. ., pp. 927-939.

21. Luo, G. Y., Osypiw, D., Irle, M., 2000, "Real-time condition monitoring by significant and natural frequencies analysis of vibration signal with wavelet filter and autocorrelation enhancement," Journal of Sound and Vibration, Vol. 263.3, pp. 413-430.

22. Yang, D.-M., Stronach, A. F., MacConnell, P., 2003, "The Application of Advanced Signal Processing Techniques to Induction Motor Bearing Condition Diagnosis," Meccanica, Vol. 38., pp. 297-308.

23. Isermann, R., 1993, "Fault Diagnosis of Machines via Parameter Estimation and Knowledge Processing," Automatica, Vol. 29.4, pp. 815-835.

24. D. Yu, 1997, "Fault Diagnosis for a Hydraulic Drive System Using a Parameter-Estimation M," Control Eng. Practice, Vol. .5, pp. 1283-1291.

25. An, L., Sepehri, N., 2003, "Hydraulic Actuator Circuit Fault Detection Using Extended Kalman Filter," Proceedings of the American Control Conference, Vol. ., pp. 4261-4266.

26. Zavarehi, M. K., Sassani, F., Lawrence, P. D., 2000, "Condition Monitoring of a Hydraulic Valve Through On-Line Estimation of the Valve Orifice Area Profile," SAE Technical Paper Series, pp. .

27. Mechefske, C. K., 1998, "Objective Machinery Fault Diagnosis Using Fuzzy Logic," Mechanical Systems and Signal Processing, Vol. 12.6, pp. 855-862.

28. Vingerhoeds, R.A., Janssens, P., Netten, B.D., Aznar Fernandez-Montesinos, M., 1995, "Enhancing Off-Line and On-Line Condition Monitoring and Fault Diagnosis," Control Engineering Practice, Vol. 3.11, pp. 1515-1528.

29. Karpenko, M., Sepehri, N., 2002, "Neural network classifiers applied to condition monitoring of a pneumatic process valve actuator," Engineering Applications of Artificial Intelligenc, Vol. 15., pp. 273-283.

Appendix A: Key Components Listing

Table A.1: Hydraulic Components

Name	Part Number	Description	Qty
Directional Valve	D1-VW1CN-JCF4	4W-3P Directional Valve; Solenoid operated, spring centered, 24V DC, 0.95A; NFPA-D03 Mounting	2
Subplate	SPD23A	Subplate Manifold, NFPA-D03 to NPTF-3/4 Side Ported	1
Accumulator	A2N0005D1K	Piston Accumulator, 2 inch bore, 5 cubic in. capacity; SAE#12 port; charged to 700 psi	1
Filter	15CN210QN50M4M41	Inline Pressure Filter, SAE#12 Port; up to 175 in^3/min	1
Check Valve	C1200B	NPTF-3/4 Ported Check Valve	1
Relief Valve	RDH082S15-4P	Variable Setting Relief Valve; NPT1/4 Port; Cartridge Valve; set pressure 1500 psi; body P/N B08-2-4P	3
Manual Valve	BVSS2203-B	Lever operated manual valve	1
Pressure Transducer	Y913	Bourdon-Sedenne Y913; 4-20mA; 0-1500psi; accuracy 2% full scale	3
Gage	PG-2000 SG25	Accutek pressure gage; 0-2000psi	
Cylinder	HC5345	Teleflex SeaStar Hydraulic Steering cylinder; Double acting; bore size 1.0 in^2	1
Pump		Pump from Teleflex; Reversible gear pump; Up port: 1400psi, Down Port: 650 psi; 200 psi 175 in^3/min at 24 V 40 A; 1000 psi 120 in^3/min at 24V 60 A	1
Reservoir		Reservoir from Teleflex; capacity approx. 1 L; use fluid HA5430	
Hydraulic Fluid	HA5430	Teleflex "Hynautic" Steering Fluid	3
Steering Hose	133744	Teleflex Steering Hoses; 12 feet long	2
Hose Assembly	F451TC-07-07-4-4-4x24	2 foot hydraulic hose, NPT 1/4-18 (F) ends; id=0.25; od=0.56; SAE100R17-4; rate pressure 300 psi	7

Table A.2: Hydraulic Fittings

Name	Part Number	Description	Qty
Adapter	0101-12-12	NPT 3/4 MM	4
Adapter	0101-4-4	NPT 1/4 MM	6
Converter	0507-12-12	NPT 3/4 (F) to SAE#12 (M)	2
Reducer	0102-6-4	NPT 3/8 (M) to NPT 1/4 (M)	2
Reducer	0102-8-4	NPT 1/2 (M) to NPT 1/4 (F)	2
Swivel Converter	0507-12-12	NPT 3/4 (F) to SAE#12 (M)	3
Swivel Reducer	0107-12-8	NPT 3/4 (M) to NPT 1/2 (F)	2
Swivel Reducer	0107-6-4	NPT 3/8 (M) to NPT 1/4 (F)	3
T-Junction	012T-12-12	NPT 3/4 MFF	2
T-Junction	012T-4-4	NPT 1/4 MFF	5
T-Junction	1/4 RRS-S	NPT 1/4 MMM	4

Table A.3: Structural Elements

Name	Part Number	Description	Qty
Bearing pillow blocks		Pillow block for the shaft	2
Linear Shaft		Thomson linear race steel shaft	1
Shaft support blocks		Thomson steel shaft support blocks	2

Table A.4: Electrical Components

Name	Part Number	Description	Qty
Motor Solid State Relay	Digikey: CC11039-ND; Crydom: D1D40	Input: 0-5 VDC; Output: 12VDC, 40A	1
Current Diode	Digikey: MUR1520IR-ND;	Diodes/RECTIFIER FAST 200V 15A TO-220AC	3
Pressure Transducers		Hydraulic Pressure transducers	3
Linear Encoder		Optical Linear Encoder	1
Accelerometer		Single Axis accelerometer	1
Terminal Strip	WM590G-ND	8.00mm Two-Screw terminal strip	4
Multi Photocoupler	PS2501-4	NEC High Isolation voltage single transistor type multi photocouplers	2
ECU	SCM40	SCM40, 200MHz, 64Bit embedded computing platform	1
Analog Input Board	TMAI01	Sitek analog input board	1
Digital Input Board	TMDI01	Sitek digital input board	1
Digital Output Board	TMDO01	Sitek digital output board	1

Appendix B: Equation Supplement

$$
\begin{aligned}
y(t) = \; & \frac{1}{\Delta^2\omega^2}\Bigg[-4ku_0+8ku_1-4ku_2+16ku_0\zeta^2-32ku_1\zeta^2+16ku_2\zeta^2 \\
& -8ktu_0\zeta\omega+16ktu_1\zeta\omega-8ktu_2\zeta\omega+6ku_0\Delta\zeta\omega-8ku_1\Delta\zeta\omega+2ku_2\Delta\zeta\omega+2kt^2u_0\omega^2 \\
& -4kt^2u_1\omega^2+2kt^2u_2\omega^2-3ktu_0\Delta\omega^2+4ktu_1\Delta\omega^2-ktu_2\Delta\omega^2+ku_0\Delta^2\omega^2 \\
& \frac{-1}{2\sqrt{-1+\zeta^2}}\Bigg(E^{t(-\zeta\omega+\sqrt{-1+\zeta^2}\omega)}\Delta^2\omega\Bigg(-dy_0-k\frac{(8u_0\zeta-16u_1\zeta+8u_2\zeta+3u_0\Delta\omega-4u_1\Delta\omega+u_2\Delta\omega)}{\Delta^2\omega} \\
& +\frac{(\zeta+\sqrt{-1+\zeta^2})(-y_0\Delta^2\omega^2+k(u_0(-4+16\zeta^2+6\Delta\zeta\omega+\Delta^2\omega^2)+2(-4u_1(-1+4\zeta^2+\Delta\zeta\omega)+u_2(-2+8\zeta^2+\Delta\zeta\omega))))}{\Delta^2\omega}\Bigg)\Bigg) \\
& +E^{t(-\zeta\omega-\sqrt{-1+\zeta^2}\omega)}\Delta^2\omega^2\Bigg(y_0-\frac{k(u_0(-4+16\zeta^2+6\Delta\zeta\omega+\Delta^2\omega^2)+2(-4u_1(-1+4\zeta^2+\Delta\zeta\omega)+u_2(-2+8\zeta^2+\Delta\zeta\omega)))}{\Delta^2\omega^2} \\
& +\frac{-dy_0-\dfrac{k(8u_0\zeta-16u_1\zeta+8u_2\zeta+3u_0\Delta\omega-4u_1\Delta\omega+u_2\Delta\omega)}{\Delta^2\omega}}{2\sqrt{-1+\zeta^2}\omega} \\
& +\frac{(\zeta+\sqrt{-1+\zeta^2})(-y_0\Delta^2\omega^2+k(u_0(-4+16\zeta^2+6\Delta\zeta\omega+\Delta^2\omega^2)+2(-4u_1(-1+4\zeta^2+\Delta\zeta\omega)+u_2(-2+8\zeta^2+\Delta\zeta\omega))))}{2\sqrt{-1+\zeta^2}\Delta^2\omega^2}\Bigg)\Bigg]
\end{aligned}
\tag{B.1}
$$

Equation B.1 (also equation 3.3) can be expressed in the form:

$$
y(t+2\Delta)=\frac{1}{\Delta^2\omega^2}(A_N+A_I+A_R)\begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ y_0 \\ dy_0 \end{bmatrix}^T
\tag{B.2}
$$

Where:

$$
A_N=\begin{bmatrix} 2k(-2+8\zeta^2-\Delta\zeta\omega) \\ 8k(1-4\zeta^2+\Delta\zeta\omega) \\ k(-4+16\zeta^2-6\Delta\zeta\omega+\Delta^2\omega^2) \\ 0 \\ 0 \end{bmatrix}
\tag{B.3}
$$

$$A_I = e^{-\Delta\zeta\omega}\left(e^{-\Delta\omega\sqrt{-1+\zeta^2}} - e^{\Delta\omega\sqrt{-1+\zeta^2}}\right)\frac{1}{2\sqrt{-1+\zeta^2}}\begin{bmatrix} -k(-3\Delta\omega+\zeta(-12+16\zeta^2+6\Delta\zeta\omega+\Delta^2\omega^2)) \\ 4k(-\Delta\omega+2\zeta(-3+4\zeta^2+\Delta\zeta\omega)) \\ -k(-\Delta\omega+2\zeta(-6+8\zeta^2+\Delta\zeta\omega)) \\ \Delta^2\zeta\omega^2 \\ \Delta^2\omega \end{bmatrix} \qquad \text{B.4}$$

$$A_R = e^{-\Delta\zeta\omega}\left(e^{-\Delta\omega\sqrt{-1+\zeta^2}} + e^{\Delta\omega\sqrt{-1+\zeta^2}}\right)\begin{bmatrix} -\frac{1}{2}k(-4+16\zeta^2+6\Delta\zeta\omega+\Delta^2\omega^2) \\ 4k(-1+4\zeta^2+\Delta\zeta\omega) \\ -k(-2+8\zeta^2+\Delta\zeta\omega) \\ \frac{1}{2}\Delta^2\omega^2 \\ 0 \end{bmatrix} \qquad \text{B.5}$$

When $\zeta>1$, equation B.4 and equation B.5 evaluate numerically as written, and easily interpreted by the simple math interpreter. However, when $\zeta<1$, equation B.4 and equation B.5 can be slightly re-arranged to replace the complex components with:

$$A_I = 2e^{-\Delta\zeta\omega}\sin(\Delta\omega\sqrt{1-\zeta^2})\frac{1}{2\sqrt{1-\zeta^2}}\begin{bmatrix} -k(-3\Delta\omega+\zeta(-12+16\zeta^2+6\Delta\zeta\omega+\Delta^2\omega^2)) \\ 4k(-\Delta\omega+2\zeta(-3+4\zeta^2+\Delta\zeta\omega)) \\ -k(-\Delta\omega+2\zeta(-6+8\zeta^2+\Delta\zeta\omega)) \\ \Delta^2\zeta\omega^2 \\ \Delta^2\omega \end{bmatrix} \qquad \text{B.6}$$

$$A_R = 2e^{-\Delta\zeta\omega}\cos(\Delta\omega\sqrt{1-\zeta^2})\begin{bmatrix} -\frac{1}{2}k(-4+16\zeta^2+6\Delta\zeta\omega+\Delta^2\omega^2) \\ 4k(-1+4\zeta^2+\Delta\zeta\omega) \\ -k(-2+8\zeta^2+\Delta\zeta\omega) \\ \frac{1}{2}\Delta^2\omega^2 \\ 0 \end{bmatrix} \qquad \text{B.7}$$

By expanding Equation B.2 with the substitutions from equations B.3, B.4, and B.5 or equations B.3, B.6, and B.7, one arrives at the conclusion in equation B.8 (also equation 3.4).

$$y(t+2\Delta) = A_0 y(t) + A_1\frac{dy(t)}{dt} + A_2 u(t) + A_3 u(t+\Delta) + A_4 u(t+2\Delta) \qquad \text{B.8}$$

Appendix C: Software Configuration

Table C.1: Health Monitor Parameters

Parameter	Value	Unit	Parameter	Value	Unit
global timestep	0	seconds	*Rate Limiter: Best*		
DHMS Parameters			rate_limit_pos	6000000	µm/s^2
update rate	1		rate_limit_neg	-6000000	µm/s^2
timestep	0	seconds	*Rate Limiter: Low*		
forecast rate	4		rate_limit_pos	5750000	µm/s^2
horizon time	0.02	seconds	rate_limit_neg	-6250000	µm/s^2
averaging horizon	30		*Rate Limiter: High*		
averaging horizon time	0.12	seconds	rate_limit_pos	6250000	µm/s^2
window average horizon	30		rate_limit_neg	-5750000	µm/s^2
dynamic compensation time	0.03	seconds	*Flow Saturater*		
Non-Linear Variable Delay			low_sat	-68000	µm/s
negdelay	0.05	seconds	high_sat	68000	µm/s
neg0delay	0.12	seconds	*Time Tolerance*		
negswitchdelay	0.07	seconds	pre_history_size	0	
posdelay	0.05	seconds	post_history_size	1	
pos0delay	0.11	seconds	*Position Sensor Parameters*		
posswitchdelay	0.07	seconds	filter cutoff	40	Hz
Quadratic Predictor			filter cutoff gain	3	
k (k)	1		filter order	3	
wn (ω_n)	70	rad/s	*Execution Parameters*		
wn_l (ω_{lb})	40	rad/s	Position Tolerance	2300	µm
wn_h (ω_{ub})	90	rad/s	Input Amplitude	26250	µm
zd (ζ)	0.5		Input Period 1	1.5	seconds
zd_l (ζ_{lb})	0.4		Input Period 2	1	seconds
zd_h (ζ_{ub})	0.8		Input Period 3	0.5	seconds
			Input Period 4	0.33	seconds

Appendix D: Software Documentation

Table D.1: DHMS Class Description

Class: DHMS Inherits from: none		
***Attributes** (Private)*		
Class: DHMS		
Name	***Type***	***Description***
iri	var (integer)	Instantaneous Response Indicator value
ari	var (integer)	Average Response Indicator value
dri	var (integer)	Dynamics Response Indicator value
yp	var (integer)	Prediction of the output state
yp_l	var (integer)	Low threshold for prediction of the output state
yp_h	var (integer)	High threshold for prediction of the output state
Ts	fvar (float)	Time step (ms) between each loop closing
^Models	Generic_linked_list (Cstring)	Linked list of all of the models Forth names used within this DHMS object
NumInputs	var (integer)	Number of inputs used
^Inputs	var (pointer to array of integer)	Array of input states used
^InputMethods	var (pointer to array of Cstring)	Array of methods to acquire each of the system inputs that are used
^InputErrors	var (pointer to array of Cstring)	Array of methods used to check the error status of each of the system inputs that are used
NumStates	var (integer)	Number of states managed
^States	var (pointer to array of integer)	Array of system states (start states)
^StateMethods	var (pointer to	Array of methods to acquire each of the system states that are

Attributes *(Private)*		
Class: DHMS		
Name	***Type***	***Description***
	array of Cstring)	used
^StateErrors	var (pointer to array of Cstring)	Array of methods used to check the error status of each of the system states that are used
^ModelExec	Cstring	Forth Word called by DHMS object to run the model Forth word is expected to have Parameters: { input1 ... inputn } and Returns: { int: yp, int: yp_l, int: yp_h } Method runs the models.
^UpdateExec	Cstring	Forth Word called by DHMS object to update the states Forth word is expected to have Parameters: { state1 ... staten } and Returns: { int: iri, int: ari, int: dri } Method runs the health routines and updates the model states with agreed sensed value (ie. The sensor value after LED methods have been run)

Methods *(Public)*	
Class: DHMS	
Name	***Description***
Get_Ts:	Parameters: { none }. Returns: { float: Ts }
Set_Ts:	Parameters: { float: Ts }. Returns: { none } Sets the new system time step, applies changes to all models in ^Models.
Reset:	Parameters: { none }. Returns: { none } Resets all of the models to their initialized state.
RunModel:	Parameters: { none }. Returns: { none } Sets the DHMS object "sensor" values
UpdateStates:	Parameters: { none }. Returns: { none } Sets the DHMS object "indicator" values
Get_Sensor:	Parameters: { none }. Returns: { int: yp, int: yp_l, int: yp_h }
Get_Health:	Parameters: { none }. Returns: { int: iri, int: ari, int: dri }
Get_IRI:	Parameters: { none }. Returns: { int: iri}
Get_ARI:	Parameters: { none }. Returns: { int: ari}
Get_DRI:	Parameters: { none }. Returns: { int: dri}
FetchInputs:	Parameters: { none }. Returns: { none } Calls all of the methods in ^InputMethods
CheckInputs:	Parameters: { none }. Returns: { none } Calls all of the methods in ^InputErrors

***Methods** (Public)* Class: DHMS	
Name	***Description***
FetchStates:	Parameters: { none }. Returns: { none } Calls all of the methods in ^StateMethods
CheckStates:	Parameters: { none }. Returns: { none } Calls all of the methods in ^StateErrors
AddInput:	Parameters: { Cstring: ^InputMethod Cstring: ^InputError }. Returns: { none }
AddState:	Parameters: { Cstring: ^StateMethod Cstring: ^StateError }. Returns: { none }
Describe:	Parameters: { none }. Returns: { none } Shows a description of the class attributes and methods
Show:	Parameters: { none }. Returns: { none } Shows the current values of class attributes
ClassInit:	Parameters: { float: Ts }. Returns: { none }

Table D.2: DHMS Transfer Function Class Description

Class: TransferF Inherits from: none		
***Attributes** (Private)*		
Class: TransferF		
Name	***Type***	***Description***
input	var (integer)	Current input to the model
state	var (integer)	Current state of the model
Ts	fvar (float)	Time step between updates
msTs	var (integer)	Time step between updates (integer)
dhmso	cstring	Forth word (string) representing the container DHMS object

***Methods** (Public)* Class: TransferF	
Name	***Description***
Reset:	Parameters: { none }. Returns: { none } Resets the transfer function to its default state
GetOutput:	Parameters: { none }. Returns: { int: value }
RunModel:	Parameters: { int: input }. Returns: { none }
UpdateStates:	Parameters: { int: state }. Returns: { none }

***Methods** (Public)*

Class: TransferF

Name	***Description***
Get_Ts:	Parameters: { none }. Returns: { float: Ts }
Set_Ts:	Parameters: { float: Ts }. Returns: { none }
Get_input:	Parameters: { none }. Returns: { int: input }
Describe:	Parameters: { none }. Returns: { none } Shows a description of the class attributes and methods
Show:	Parameters: { none }. Returns: { none } Shows the current values of class attributes
ClassInit:	Parameters: { int: start_output, cstring: dhmso }. Returns: { none }

Table D.3: DHMS-2ODE Class Description

Class: DHMS-ODE

Inherits from: TransferF

***Attributes** (Private)*

Class: DHMS-2ODE

Name	***Type***	***Description***
k	fvar (float)	System gain
wn	fvar (float)	Undamped natural frequency
wnl	fvar (float)	Low threshold of undamped natural frequency
wnh	fvar (float)	High threshold of undamped natural frequency
zd	fvar (float)	Damping ratio
zdl	fvar (float)	Low threshold of damping ratio
zdh	fvar (float)	High threshold of damping ratio
delta	fvar (float)	Total prediction time
yp	fvar (float)	Current model output prediction for system state
yp_h	fvar (float)	Current model output estimate for high threshold
yp_l	fvar (float)	Current model output estimate for low threshold
^yps	var (pointer to array of float)	Array of 5 elements, containing the current predictions of system output thresholds
forecast_rate	var (integer)	Number of periods to forecast over
history_size	var (integer)	Number of periods currently stored in history_x arrays

***Attributes** (Private)*

Class: DHMS-2ODE

Name	***Type***	***Description***
history_start	var (integer)	Current index of the start point in the history_x arrays
history_mid	var (integer)	Current index of the mid point in the history_x arrays
history_end	var (integer)	Current index of the end point in the history_x arrays
^history_u	var (pointer to array of float)	Array with forecast_rate elements storing a history of input states
^history_y	var (pointer to array of float)	Array with forecast_rate elements storing a history of system states
^history_dy	var (pointer to array of float)	Array with forecast_rate elements storing a history of system states (derivative)
^temp_A	var (pointer to array of float)	Temporary Calculation buffer
^A	var (pointer to array of float)	Array of values used to optimize the calculation of prediction at each step. See Equation B.8 (Also Equation 3.4). Used with parameters: w=wn, z=zd, k=k
^A_ll	var (pointer to array of float)	Used with parameters: w=wnl, z=zdl, k=k
^A_hl	var (pointer to array of float)	Used with parameters: w=wnh, z=zdl, k=k
^A_lh	var (pointer to array of float)	Used with parameters: w=wnl, z=zdh, k=k
^A_hh	var (pointer to array of float)	Used with parameters: w=wnh, z=zdh, k=k
^A_mm	var (pointer to array of float)	Calculated at closing time for max/min point.
^C	var (pointer to array of float)	Buffer used to calculate w where dw/dt=0
^D	var (pointer to array of float)	Buffer used to calculate z where dz/dt=0

***Methods** (Public)*

Class: DHMS-2ODE

Name	***Description***
RunModel:	Parameters: { int: input }. Returns: { none }
UpdateStates:	Parameters: { int: state, int: state_derivative }. Returns: { none }

***Methods** (Public)*

Class: DHMS-2ODE

Name	***Description***
GetOutput:	Parameters: { none }. Returns: { int: yp, int: yp_l, int: yp_h }
get_yp:	Parameters: { none }. Returns: { int: yp }
get_yp_l:	Parameters: { none }. Returns: { int: yp_l }
get_yp_h:	Parameters: { none }. Returns: { int: yp_h }
get_history_start:	Parameters: { none }. Returns: { int: history_start }
get_history_mid:	Parameters: { none }. Returns: { int: history_mid }
get_history_end:	Parameters: { none }. Returns: { int: history_end }
get_history_u:	Parameters: { int: index }. Returns: { int: history_u[index] }
get_history_y:	Parameters: { int: index }. Returns: { int: history_y[index] }
get_history_dy:	Parameters: { int: index }. Returns: { int: history_dy[index] }
Describe:	Parameters: { none }. Returns: { none } Shows a description of the class attributes and methods
Show:	Parameters: { none }. Returns: { none } Shows the current values of class attributes
ClassInit:	Parameters: { int: forecast_rate, int: start_output, cstring: dhmso, float: k, float: wn, float: wnl, float: wnh, float: zd, float: zdl, float: zdh }. Returns: { none }

Table D.4: DelayRelay Class Description

Class: DelayRelay

Inherits from: TransferF

***Attributes** (Private)*

Class: DelayRelay

Name	***Type***	***Description***
negdelay	fvar (float)	delay when dropping from 0 to -1 (s)
neg0delay	fvar (float)	delay when zeroing from -1 to 0 (s)
posdelay	fvar (float)	delay when rising from 0 to 1 (s)
pos0delay	fvar (float)	delay when zeroing from 1 to 0 (s)
negswitchdelay	fvar (float)	delay when zeroing from -1 to 0 powered (s)
posswitchdelay	fvar (float)	delay when zeroing from 1 to 0 powered (s)
sum	var (integer)	current internal state of input parts

***Attributes** (Private)*		
Class: DelayRelay		
Name	***Type***	***Description***
ratepos1	var (integer)	rate of change of internal state for rise to +1
ratepos0	var (integer)	rate of change of internal state for zero from +1
rateneg1	var (integer)	rate of change of internal state for fall to -1
rateneg0	var (integer)	rate of change of internal state for zero from -1
ratepos0p	var (integer)	rate of change of internal state for zero from +1 powered
rateneg0p	var (integer)	rate of change of internal state for zero from -1 powered

***Methods** (Public)*	
Class: DelayRelay	
Name	***Description***
RunModel:	Parameters: { int: input }. Returns: { none }
UpdateStates:	Parameters: { int: state }. Returns: { none } Overrides the internal system state
GetOutput:	Parameters: { none }. Returns: { int: state }
Describe:	Parameters: { none }. Returns: { none } Shows a description of the class attributes and methods
Show:	Parameters: { none }. Returns: { none } Shows the current values of class attributes
ClassInit:	Parameters: { float: negdelay, float: neg0delay, float: negswitchdelay, float: posdelay, float: pos0delay, float: posswitchdelay }. Returns: { none }

Table D.5: RateLimiter Class Description

Class: RateLimiter		
Inherits from: TransferF		
***Attributes** (Private)*		
Class: RateLimiter		
Name	***Type***	***Description***
rl_pos	fvar (float)	positive rate limit
rl_neg	fvar (float)	negative rate limit
forecast_rate	var (integer)	number of periods to forecast over
Delta	fvar (float)	prediction time
dy_pos	var (integer)	max change positive

Attributes *(Private)*

Class: RateLimiter

Name	***Type***	***Description***
dy_neg	var (integer)	max change negative

Methods *(Public)*

Class: RateLimiter

Name	***Description***
RunModel:	Parameters: { int: input }. Returns: { none }
UpdateStates:	Parameters: { int: state }. Returns: { none } Sets the start state for the next iteration
GetOutput:	Parameters: { none }. Returns: { int: state }
Describe:	Parameters: { none }. Returns: { none } Shows a description of the class attributes and methods
Show:	Parameters: { none }. Returns: { none } Shows the current values of class attributes
ClassInit:	Parameters: { int: forecast_rate, float: rate_limit_pos, float: rate_limit_neg }. Returns: { none }

Table D.6: Relay Class Description

Class: Relay

Inherits from: TransferF

Attributes *(Private)*

Class: Relay

Name	***Type***	***Description***
low_out	var (integer)	low output
high_out	var (integer)	high output
drop_threshold	var (integer)	threshold when rise from low to high
rise_threshold	var (integer)	threshold when drop from high to low

Methods *(Public)*

Class: Relay

Name	***Description***
RunModel:	Parameters: { int: input }. Returns: { none }
UpdateStates:	Parameters: { int: state }. Returns: { none }

***Methods** (Public)*

Class: Relay

Name	***Description***
	Updates the input without updating the output
GetOutput:	Parameters: { none }. Returns: { int: state }
Describe:	Parameters: { none }. Returns: { none } Shows a description of the class attributes and methods
Show:	Parameters: { none }. Returns: { none } Shows the current values of class attributes
ClassInit:	Parameters: { int: low_out, int: high_out, int: drop_threshold, int: rise_threshold }. Returns: { none }

Table D.7: Saturation Class Description

Class: Saturation

Inherits from: TransferF

***Attributes** (Private)*

Class: Saturation

Name	***Type***	***Description***
low_sat	var (integer)	low output
high_sat	var (integer)	high output
sat_state	var (flags)	integer flags use state each of the saturation points

***Methods** (Public)*

Class: Saturation

Name	***Description***
RunModel:	Parameters: { int: input }. Returns: { none }
UpdateStates:	Parameters: { int: state }. Returns: { none } Updates the input without updating the output
GetOutput:	Parameters: { none }. Returns: { int: state }
Describe:	Parameters: { none }. Returns: { none } Shows a description of the class attributes and methods
Show:	Parameters: { none }. Returns: { none } Shows the current values of class attributes
ClassInit:	Parameters: { int: use_low, int: use_high, int: low_sat, int: high_sat }. Returns: { none }

Table D.8: TimeTolerance Class Description

Class: TimeTolerance

Inherits from: TransferF

***Attributes** (Private)*

Class: TimeTolerance

Name	***Type***	***Description***
pre_history_size	var (integer)	Number of history points to consider in tolerance before current state time
post_history_size	var (integer)	Number of history points to consider in tolerance after current state time
history_size	var (integer)	Total size of the history (pre + post)
^u_h	var (pointer to array of integer)	history of high inputs
^u_l	var (pointer to array of integer)	history of low inputs
^y	var (pointer to array of integer)	history of values
newest_y	var (integer)	current index of y
oldest_y	var (integer)	index of earliest data point
newest_u	var (integer)	current index of u
oldest_u	var (integer)	index of earliest input point
used_size	var (integer)	amount of history recorded

***Methods** (Public)*

Class: TimeTolerance

Name	***Description***
RunModel:	Parameters: { int: u_h, int: u_l, int: y }. Returns: { none }
UpdateStates:	Parameters: { none }. Returns: { none } does nothing
GetOutput:	Parameters: { none }. Returns: { int: yp_h, int: yp_l }
flag_output:	Parameters: { none }. Returns: { int: flag } Returns an integer flag where each bit describes the pass/fail condition of the state within the boundaries for each point in the history
get_u_h:	Parameters: { int: index }. Returns: { int: ^u_h[index] }
get_u_l:	Parameters: { int: index }. Returns: { int: ^u_l[index] }

Methods *(Public)*

Class: TimeTolerance

Name	***Description***
get_y:	Parameters: { int: index }. Returns: { int: ^y[index] }
Describe:	Parameters: { none }. Returns: { none } Shows a description of the class attributes and methods
Show:	Parameters: { none }. Returns: { none } Shows the current values of class attributes
ClassInit:	Parameters: { int: pre_history_size, int: post_history_size }. Returns: { none }

Table D.9: TransDelay Class Description

Class: TransDelay

Inherits from: TransferF

Attributes *(Private)*

Class: TransDelay

Name	***Type***	***Description***
delay	var (integer)	number of time steps to delay
history_start_in dex	var (integer)	current index of start
^y	var (pointer to array of integer)	history of state increments

Methods *(Public)*

Class: TransDelay

Name	***Description***
RunModel:	Parameters: { int: input }. Returns: { none }
UpdateStates:	Parameters: { int: state }. Returns: { none } Sets the current history of states all to the new state
GetOutput:	Parameters: { none }. Returns: { int: state }
Describe:	Parameters: { none }. Returns: { none } Shows a description of the class attributes and methods
Show:	Parameters: { none }. Returns: { none } Shows the current values of class attributes
ClassInit:	Parameters: { int: delay }. Returns: { none }

Table D.10: WindowAvg Class Description

Class: WindowAvg

Inherits from: TransferF

***Attributes** (Private)*

Class: WindowAvg

Name	***Type***	***Description***
steps	var (integer)	number of time steps to run the average over
history_start_index	var (integer)	current index of start
^y	var (pointer to array of integer)	history of state increments
sum	var (integer)	current sum of system states

***Methods** (Public)*

Class: WindowAvg

Name	***Description***
RunModel:	Parameters: { int: input }. Returns: { none }
UpdateStates:	Parameters: { none }. Returns: { none } does nothing
GetOutput:	Parameters: { none }. Returns: { int: state }
Describe:	Parameters: { none }. Returns: { none } Shows a description of the class attributes and methods
Show:	Parameters: { none }. Returns: { none } Shows the current values of class attributes
ClassInit:	Parameters: { int: steps }. Returns: { none }