Developing a Generic Architecture for Autonomic Fault Handling in Mobile Robots

Martin Doran

A dissertation submitted in partial fulfilment

of the requirements for the degree of

Doctor of Philosophy

of

Ulster University.

Department of Computing and Engineering

Ulster University

April 14, 2020

I, Martin Doran, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

The Autonomic Computing paradigm was first presented almost 18 years ago as a 20-30 year long research agenda. Organizations like NASA, have explored the possibilities of using autonomic systems in their future missions due to vast distances experienced in space exploration. Planetary and mobile robots operate in hostile environments and because of their remote location, human intervention for repairs is not possible. Hardware devices, like mobile robots, are susceptible to internal and external environmental changes, which can lead to faults occurring. Some research has been conducted in terms of handling faults in mobile robots but there is no generic autonomic model that can be used for any type of system fault, in any type of mobile robot. This Thesis describes a generic autonomic architecture for use in developing systems for managing hardware faults in mobile robots. Using autonomic principles, this Thesis focuses on how to detect faults within a mobile robot and how specialised algorithms can be deployed to compensate for the faults discovered. The initial design of a generic architecture is developed using inspiration from the MAPE-K and IMD architectures. Case studies are presented that show three different fault scenarios that can occur within the effectors, sensors and power units of a mobile robot. The results from each of the Case Studies is used to create and refine a generic autonomic architecture that can be utilized for any general mobile robot setup for fault handling. A further Case Study is presented, which exercises the generic autonomic architecture in order to demonstrate its utility. This Thesis addresses the fundamental challenges in operating remote mobile robots with little or no Human intervention. If a fault does occur within the mobile robot during field operations, then having a *self-managing* strategy as part of its pro-

cesses may result in the mobile robot continuing to function at a productive level. Research in this Thesis has provided insights into the shortcomings of existing robot design which is also discussed.

Acknowledgements

I would like this opportunity to thank my supervisors Roy Sterritt and George Wilkie, for their guidance, valuable suggestions and continuous encouragement over the last six years.

For my family and friends.

For Allstate NI for allowing me every other Monday off to work on my PhD.

Note on Access to Contents

I hereby declare that with effect from the date on which the thesis is deposited in Research Student Administration of Ulster University, I permit

1. the Librarian of the University to allow the thesis to be copied in whole or in part without reference to me on the understanding that such authority applies to the provision of single copies made for study purposes or for inclusion within the stock of another library.

2. the thesis to be made available through the Ulster Institutional Repository and/or EThOS under the terms of the Ulster eTheses Deposit Agreement which I have signed.

Contents

List of Figures

List of Tables

List of Equations

Acronyms

AUTONOMIC COMPUTING (AC)

AUTONOMIC COMPUTING INITIATIVE (ACI)

ARTIFICIAL INTELLIGENCE (AI)

AUTONOMIC ELEMENT (AE)

AUTONOMIC FAULT-MANAGEMENT (AFM)

AUTONOMIC INTELLIGENT FAULT HANDLING (AIFH)

CONCURRENCY CO-ORDINATION RUN-TIME (CCR)

DEPTH OF CHARGE (DOD)

INTELLIGENT MACHINE DESIGN (IMD)

MONITOR ANALYSE PLAN EXECUTE KNOWLEDGE (MP)

MICROSOFT ROBOTICS DEVELOPMENT STUDIO (MRDS)

ORGANIC COMPUTING (OC)

ORGANIC ROBOTIC CONTROL ARCHITECTURE (ORCA)

PROPORTIONAL INTEGRATED DERIVATIVE (PID)

SMART BATTERY SYSTEM (SBC)

SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC)

Chapter 1

Introduction

This chapter introduces the research topic of this thesis. Section 1.1 discusses the context within which the research problems are recognized. Section 1.2 discusses the problem statement. Section 1.3 presents the overall aim and objectives of this research. Finally, Section 1.4 presents the Chapter outline.

1.1 Research Context

In the late 1960's the world's first mobile robot 'Shakey' was developed by researcher's at Stanford Research Institution (now SRI International) [18]. The components attached to 'Shakey', i.e., *range finders*, *vision camera* and *bumper sensors*, are typical of components you would find on present-day mobile robots. In the 21st century, mobile robots can be found in factories, warehouses, laboratories and office spaces. These modern-day mobile robots are not just the proverbial 'pack mule' but contain on-board intelligence and real-time adaptive capabilities [19]. However, it is in the *space* industry that mobile robots are pushed to their limits in terms of reliability and robustness. These types of mobile robots or planetary rovers, operate in hostile environments; and because they are in a remote location, human intervention to make any repairs is impossible.

When a fault occurs in a mobile robot, there are various ways the operator can become aware that there is a problem. The mobile robot may sound an alarm or send a message to main operations. Other scenarios may include the mobile robot becoming motionless or perhaps veering off from its normal operating area. In any

event, human operators are there to examine, fix or replace any faulty component found on the mobile robot. If a fault occurs in a component on a planetary rover, then the robot may lose a degree of functionality depending on the extent of the fault. Human intervention is limited to 'software updates' or shutting down some on-board systems in-order to preserve other components. Each component can be subjected to various fault scenarios, i.e. intermittent, permanent or non-functioning (total failure). However, maintenance cannot be employed, and parts cannot be replaced or repaired [20]. It is therefore important that the planetary rover can adapt to new situations if an error occurs. Authors in [21], explained that faults that occurred in NASA's Spirit Rover, where due to unknown factors or environmental interactions. Fault tolerance strategies where limited, as faults in the motor system could not be resolved due to the performance degradation imposed when the Spirit Rover switched to a *system fault mode*. In 2018 NASA reported issues with the Opportunity Rover in [22]. They reported three main faults *low-power fault*, *clock fault* and *uploss fault*. When 'Opportunity' experienced a problem, it can go into so called "fault mode" where it automatically takes action to maintain its health. However, NASA could not resolve the issues with 'Opportunity', and it was declared 'lost' in February 2019.

The term 'Autonomic' is described as *occurring involuntarily or spontaneously, resulting from internal stimuli* [23][24]. The *Autonomic Computing Paradigm* is a concept that has been inspired by the human autonomic nervous system. Its main objective is that computer and software systems can manage themselves with only high-level guidance from humans [25]. One of the main aspects of autonomic computing described in [26], is that an autonomic system must perform something comparable to *healing*, it must be able to recover from routine or exceptional events that might cause some individual parts to malfunction. In the case of mobile robots (planetary rovers), operating in remote destinations, [27][28] have described the need for future NASA missions to be autonomic by necessity, owing to the complex nature of the software and hardware systems involved.

1.2 Research Problem

In the last decade, several research approaches have been taken in fault handling for mobile robots. A representative list of such approaches can be found in research such as [29][30][31]. These approaches utilize *fault tolerance* and FDI (Fault Detection and Isolation) techniques to detect, analyse and recover from hardware faults in mobile robots. Although these techniques provide fault-handling solutions in mobile robots, they are constrained by the fact that early design decisions must be applied to fault tolerance systems. This results in a lack of flexibility if roles and strategies need to be altered when the system is operating in the field. Autonomic fault handling offers *self-adaption* and the ability to use a *generic* software approach when designing a system that can react to behavioural changes [32].

In 2003, IBM presented a white paper that described an autonomic reference model in the form of the MAPE-K (Monitor, Analyses, Plan, Execute - Knowledge) feedback loop [1]. The MAPE-K architecture assumes the availability of *sensors* and *effectors* in order to gather data for managed resources. Within its *feedback* loop, there is functionality to *monitor* the sensed data, to *analyse* the current state, to *plan* corrective actions and to *execute* these actions via the *effectors*. The MAPE-K is a popular model when building *self-adaption* into current systems but its original form is normally altered, so that researchers can adapt it to their own specific requirements [33][34][12][13]. It therefore begs the question whether the MAPE-K model, in its original configuration [1], goes far enough for developers to design an effective *self-adaptive* solution? Research reported in [35], proposed that there are sources of *uncertainty* in the MAPE-K which may affect its components. The MAPE-K *monitoring* component may be affected by inherent imperfections of sensors. These imperfections may be a result of *sensing latency*: by the time analyses occurs the information is outdated. Other imperfections could be a result of *sensor inaccuracy*: the sensor cannot distinguish deviations between the actual reading and the ideal reading. The authors in [36], state that within the MAPE-K the role of *knowledge* is unclear. In their Formal Reference Model (FORMS), they elaborate on *knowledge* by making distinctions between (1) architectural model (struc-

tural and behavioural data); (2) goal model (action policies and utility functions); (3) environmental model (in which the managed autonomic element is situated). In robotics, research reported in [4][5], discusses the IMD (Intelligent Machine Design) as a model where computing systems should be designed to operate autonomously, unattended and be highly robust. The IMD consists of three levels with 'intelligence' increasing from the lower to the higher level. The lowest level (Reaction), is in direct contact with sensors and motor elements. The highest level (Reflection), makes decisions based on knowledge and experiences. Research in [6], makes a case for integrating the autonomic MAPE-K model to operate on each level of the IMD architecture. But does the IMD design go far enough for the development of *self-adaption* in robots? Is there a case for using both MAPE-K and IMD to create a *hybrid* architecture for fault handling in mobile robots?

1.3 Research Objectives

The goal of this research is to develop a generic autonomic architecture for fault handling in mobile robots. Using the autonomic MAPE-K and IMD models as a basis, the generic autonomic architecture is gradually developed through case study interrogation. The motivation is that developers can use the generic autonomic architecture as a reference when designing future mobile robotic systems. The proposed approach is intended to retain the important elements found in *autonomic computing* but also to improve upon current *autonomic* models.

The following are the set of objectives:

1. Research the main attributes associated with the MAPE-K and IMD models and establish if they do enough within the autonomic robotic community or is there areas that can be improved upon.

2. Develop *self-awareness* (i.e. becoming aware), when establishing a fault scenario within a mobile robot.

3. Using case study methods, identify common design patterns in-order to develop an autonomic generic architecture for handling component faults in mobile robots.

1.4 Chapter Outline

The structure of this thesis is as follows:

Chapter 1 - Introduction.

Chapter 2 - Literature Review: the origins of Autonomic Computing are discussed and why its introduction is an attempt to address the growing complexity of computer systems we find today in modern computing. An alternative model is also discussed in the form of the Intelligent Machine Design (IMD) and how it can be integrated with IBM's MAPE-K architecture. Further discussions are centred around Organic Computing and how it differs from the original autonomic computing concept to form its own architectural development. This chapter also describes fault handling in mobile robots and how autonomic systems compare to traditional fault tolerance systems. Finally, current research in the autonomic computing field is discussed and how researchers have used the autonomic MAPE-K and the robotic IMD in their work.

Chapter 3 - Research Hypothesis and Overview: in this chapter the thesis goals are presented and how design patterns can be used in analysis of faults in robots. The Research Method is also presented in this chapter showing how case study Methodology is used as a means to develop the autonomic generic architecture. Further discussions in this chapter explain how the MAPE-K and IMD models can be integrated to form the basis of a new architecture. Finally, SDLC (Software Development Life Cycle), is discussed as the methodology used within the case studies of this thesis.

Chapter 4 - Self-Adaptive Mobile Robot Wheel Alignment - case study: this chapter demonstrates how an *autonomic management system* can be implemented to handle a 'wheel alignment' fault in a mobile robot. The SDLC model is used to analyse, design, implement and test each stage of the case study. This chapter also presents the initial development of the autonomic generic architecture and how it can be designed to handle a 'wheel alignment' fault in a mobile robot. Finally, testing and evaluation methods are used to demonstrate the *compensation policy* applied to the 'wheel alignment' fault and how it improves its performance over

distance and time.

Chapter 5 - Autonomic Sonar Sensor Fault Management for Mobile Robots - case study: in this chapter, methods are discussed for how to handle *sonar sensor* faults in a mobile robot. The SDLC methodology is used for the analysis, implementation and testing of the *sonar sensor* fault. The autonomic generic architecture is further developed by introducing *Knowledge Base* attributes to identify, analyse and finally, compensate for the fault. The chapter concludes with an evaluation process to test the *compensation policy* and how it affects the performance of the robot in terms of detecting objects.

Chapter 6 - Autonomic Management for Mobile Robot Battery Degradation - case study: in this chapter, methods are discussed on how to handle battery degradation within a mobile robot. This case study uses SDLC methods for designing, implementing and testing. The autonomic generic architecture is further developed by incorporating control loops (reactive and proactive), in identifying the fault (awareness), processing the fault (analysis) and compensating for the fault (adjustment). The chapter concludes with testing the *compensation policy* and how it affects the performance of the mobile robot in relation to power consumption and task completion.

Chapter 7 - Generic Architecture for Fault Management (AIFH): in this chapter the findings accumulated from the case studies (Chapters 4, 5 and 6), are used to design a fully fledged generic autonomic architecture for handling faults in a mobile robot. A *High-level* design is introduced, explaining the elements found in the Autonomic and System Managers. This chapter then explorers the *self-healing* module (the module that handles the feed-back control loops) and the *knowledge-base module* (which includes attributes such as policies, tolerance values and historical data). This chapter then presents the *low-level* design of the AIFH (Autonomic Intelligent Fault Handling) architecture. Each layer of the AIFH architecture (Awareness, Analysis and Adjustment) is discussed, and how those layers play a role in detecting, processing and compensating for faults. Finally, in this chapter, the generic autonomic architecture (AIFH) is evaluated by introducing a further case study (Stereo

Camera Vision Fault). The main goal in this Section is to demonstrate the utility of the generic architecture in a new *fault* scenario.

Chapter 8 - Conclusions and Future Work: provides conclusions to the research discussed in this thesis and what future work is under proposed.

1.5 Overview of Publications

The research presented in this thesis produced six publications and one submission. These publications and submission, span over the six years of part-time study.

The **first** publication was presented and published for the 11th IEEE International Conference and Workshops on the Engineering of Autonomic and Autonomous Systems [15] (2014). Using indoor GPS, a mobile robot was tracked to discover wheel alignment faults. This paper introduced to the research, an autonomic strategy to compensate for the wheel alignment fault. This publication contributed to research carried out in Chapter 4.

The **second** publication was presented and published for the 13th Symposium on Advanced Space Technologies in Robotics and Automation [37] (2015). This publication introduces a high-level concept of robotics based on the autonomic computing paradigm, with the aim of developing *self-managing* and autonomous software for space missions. The three main areas of research included *autonomic architecture*, *cooperation between robots* and *development of middle-ware*.

The **third** publication was presented and published for the Eighth International Conference on Adaptive and Self-Adaptive Systems and Applications [38] (2016). This publication detailed how an Autonomic Management System (based on the MAPE-K [1]), could be applied for detecting, evaluating and compensating a wheel alignment fault on a mobile robot. This publication relates to research carried out in Chapter 4.

The **fourth** publication was presented and published for the 19th International Conference on Autonomic Computing and Computer Engineering, [39] (2017). In this publication, an extension of the MAPE-K architecture is introduced as the AAA (Awareness), (Analysis) and (Adjustment) model, to handle sonar sensor faults in

mobile robots. This publication relates to research carried out in Chapter 5.

The **fifth** publication was presented and published for the 20th International Conference on Autonomic Computing and Computer Engineering, [40] (2018). In this publication, research was conducted on how battery degradation can affect task performance on a mobile robot. Autonomic principles where applied to handle the battery degradation fault, so that the mobile robot could still operate in some capacity. This publication relates to research carried out in Chapter 6.

The **sixth** publication was presented and published for SMC-IT (International Conference on Space Mission Challenges for Information Technology), IEEE/NASA, in Pasadena, California (2019). In this publication, the AIFH architecture was presented as extension to the MAPE-K and IMD models and how it could be implemented for fault handling in mobile robots.

Finally, the **seventh** paper was published by Innovations in System and Software Engineering, NASA Journal, (2020). This journal represents a summary of the research completed by the author over the duration of the PhD. Journal title, 'Autonomic Architecture for Fault Handling in Mobile Robots'.

Chapter 2

Literature Review

2.1 Introduction

The following literature review concerns itself with origins of Autonomic Computing and how it can influence how we develop computers systems now and in the future. The review further explorers how hardware faults in systems can be handled using an *autonomic* approach and the importance of an *generic architecture* is in delivering these goals. This Chapter is organized as follows: Section 2.2 deals with the fundamentals in Autonomic Computing, the types of attributes and various interpretations that have been employed by researchers in developing *autonomic* models. Section 2.3 deals with an Autonomic approach to handling faults in mobile robots and the influence of Organic Computing in this area of research. Section 2.4 looks at how a *generic* architecture based on current autonomic models can be adapted to fault handling in computing systems. Section 2.5 contains a summary of the literature review.

2.2 Autonomic Computing

2.2.1 Origins and Motivations

The Autonomic Computing Initiative (ACI), was promoted by IBM in 2001, to address the ever-increasing complexity of computer systems [41][42]. Technological advances in computer and software systems have resulted in legacy systems being constantly updated to a point where management of these systems is becoming un-

tenable. To further the complexity, there is increased need to distribute data, applications and system resources across international and national business boundaries [43]. However, updating current systems in terms of self-* management properties [41], is a difficult task because of the dependencies and limitations that are tied to the system's earlier design strategies [44].

IBM's ACI, is inspired by the human body's nervous system. The nervous system acts and reacts to stimuli which are independent to the individual's conscious input. Just as the human body functions without interference from the individual (e.g., temperature rises and falls, breathing rate etc), the autonomic computing environment operates and reacts in response to the information it retrieves [45]. However, there are important distinctions between the autonomic activities within the human body and the activities performed in an autonomic computer system. Many of the decisions made within the human body regarding autonomic activity are involuntary, whereas decision making within a autonomic computer system, is often carried out by a Autonomic Manager which in-turn, will communicate with the System Manager to carry out tasks and the polices [46][47].

IBM proposed self-management, where functions could be accomplished by taking appropriate action based in the current state of the surrounding environment. Computer systems with self-managing components could potentially reduce the cost of owning and operating such systems. These attributes are defined as follows [1]:

- *Self-configuration* - the ability to dynamically adapt to changing environments. when changes occur, policies can be enforced that result components being deployed or removed (with minimal human intervention). Dynamic adaption can increase both growth and flexibility within an IT organization, when faced with future changes.

- *Self-healing* - has the ability to detect improper operations. It can analyse and make changes when faced with disruptions. The components operating within a *self-healing* environment, can deploy policies that can correct the computer system without affecting the surrounding environment. If components fail,

a *self-healing* system can initiate corrective action to adapt to environmental changes.

- *Self-optimizing* - the ability of the IT environment to maximize resource allocation and utilization when faced with dynamically changing workloads. Self-optimizing components can learn from experience and thus make changes to themselves in-order to achieve the overall business objectives. Self-optimization helps provide a high standard of service for Users and business clients.

- *Self-protecting* - system components have the ability to detect hostile threats and take corrective action to make themselves less vulnerable to attacks. Self-protecting capabilities allows businesses to enforce security policies that will ultimately affect productivity and customer satisfaction. Self-protection is also about monitoring the system as a whole and reacting to any changes that could affect its integrity.

2.2.2 Autonomic Model

According to IBM [48], the Autonomic attributes, referred to as self-CHOP (Configure, Heal, Optimize and Protect), are the foundations for self-managing systems. These autonomic concepts can be orchestrated using an Autonomic Manager [47], in which the Autonomic Manager implements a feedback loop. The autonomic feedback loop is integrated into the MAPE-K model (Monitor, Analyse, Plan, Execute - using a Knowledge base) - see Figure 2.1.

The MAPE architecture divides the feedback loop into four separate parts. Each part has access to the *knowledge* resource.

- *Monitor* - processes information from managed resources (sensors etc). The monitor function processes the data until it recognizes a symptom that needs to be further investigated. The autonomic system has two types of monitoring: passive and active monitoring. Passive monitoring involves using a third party monitoring function. For example, in a LINUX, VMSTAT is used to monitor CPU utilization. The Autonomic Manager can directly access third party

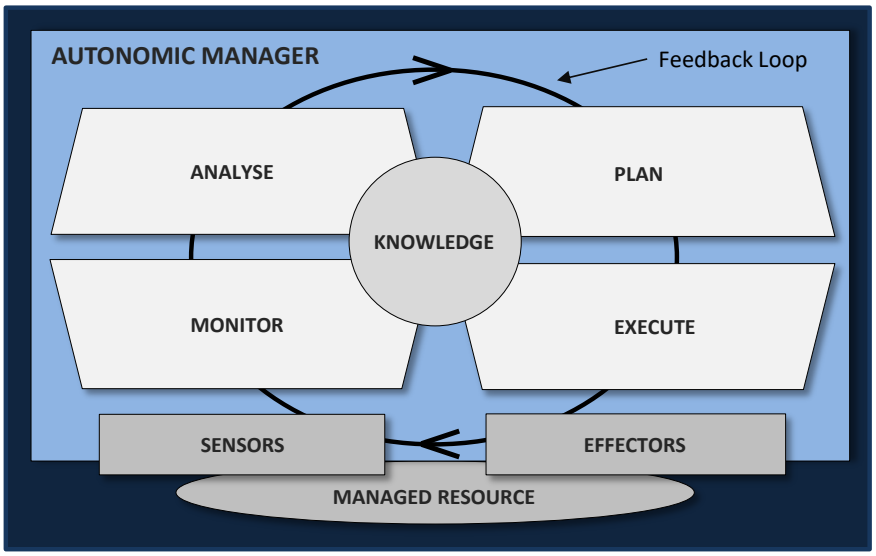


Figure 2.1: Autonomic MAPE-K model proposed by IBM [1].

monitoring tools [49]. Active monitoring means the Autonomic Manager has a direct connection with its surrounding environment, in particular with sensors. The active monitoring process can be affected by the goals set by Administration. These goals can allow the monitoring process to transform the data selected using operations such as analyzing, filtering and aggregation.

- *Analyse* - takes the symptoms identified by the *monitoring* process and performs data analysis. The analyses process draws on the *knowledge store* to make comparisons and assist with decision making. The *knowledge store* can contain *policies*, which can be implemented to investigate symptoms passed down from the *monitoring* function. The analyse function is responsible for determining the extent of the fault and if required, make a decision if changes are required.

- *Plan* - takes the symptom data from the analyse function and determines what action is needed to compensate for the fault. The *Plan* function can utilize policies that allow it to initiate an alteration within *managed* resource. A plan of action can either be static or dynamic. A *static* plan would consist of a set of steps that must be carried out in sequence, when a particular condition

has occurred. A *dynamic* plan would require choosing the best plan (from an existing collection of plans), by iterating through various *fault* scenarios. Therefore, the *Plan* function is making a hypothesis on the effect a *planned* action might take.

- *Execute* - provides the managed resource to execute the policies that where instigated by the *Plan* function. This is achieved by using the appropriate effectors. The *Execute* function has no ability to modify the surrounding environment.

Although *knowledge* is shared by all the 'parts' within the Autonomic Manager, Monitor and Analyse use the *knowledge store* extensively in order to make decisions regarding the discovery of symptoms or faults and how best to deal with them. In an autonomic computing system, *knowledge* captures data, renders it in a standard way so that it can be used by the Autonomic Manager, and that in turn enables new *knowledge* to be discovered and learned [50].

2.2.2.1 Feedback Loop Application

In the MAPE-K, the feedback loop orchestrates the interaction between the different attributes within the Autonomic Manager. The feedback loop can be described as a *closed* loop system. The feedback loop can determine whether an environmental change requires the need for *adaption* or not [51]. The autonomic systems use feedback loops to achieve self-management [52].

Research conducted in [53], shows how the MAPE-K feedback can be applied in *adaptive* systems as three distinct categories.

Independent loop - in this category, the agents possess all the main attributes within the MAPE-K. If changes occur to the local environment, then agent will become aware of these changes. However, these types of independent feedback loops are not usually found in a collective system.

Interacting loop - within a collective system, there are many interactions between agents that possess autonomic *self-properties*. Even though the MAPE-K loop exists within each agent, there is interaction between agents as each loop is

executed.

Distributed loop - as typical distributed system can contain different types of agents. These types of agents may include intelligent controllers or agents. Some distributed systems may also contain human operators.

Research reported in [25], describes the autonomic element as having a *local loop* and a *global loop*. The *local loop* is only aware of environment states that results from knowledge base that is embedded within the element. However, the *local loop* is ignorant of the overall behaviour of the entire system and therefore cannot affect any global changes. The *global loop* can handle unknown environmental states which can manifest as artificial intelligence, machine learning and human interaction. The *global loop* can initiate changes in behaviour, which can be triggered by a fall in the overall system performance levels.

Real autonomic systems require multiple management feedback loops. In research conducted by [2], the autonomic MAPE (plan/execute) is replaced with a *transition function*, which in-turn triggers the actuators - see Figure 2.2..

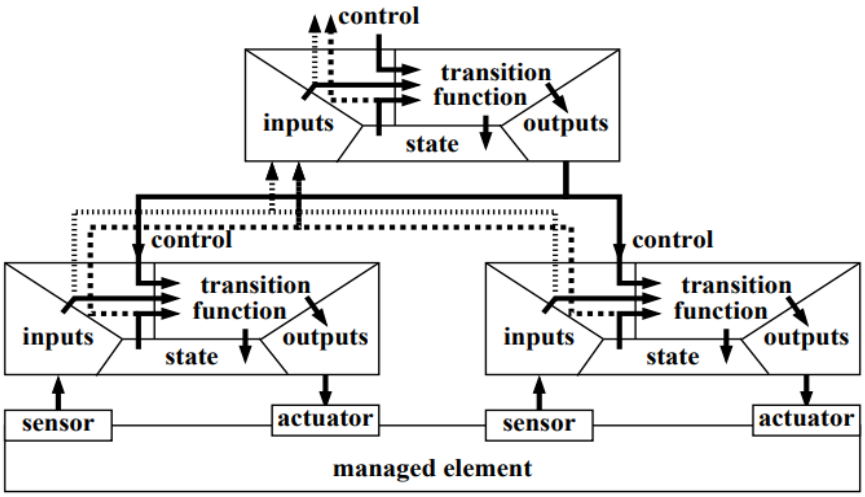


Figure 2.2: Autonomic coordination for feedback loops [2].

These types of multiple loops can be arranged in an hierarchical framework where the top layer AM (Autonomic Manager) coordinates the lower-level AM's. The hierarchical system allows each feedback loop to perform its own function-

ality but at the same time coordination of strategy and polices are enforced using controlled interfaces.

2.2.2.2 Autonomic Policies

In autonomic computing, policies play an important role as they describe how the system is guided in terms of decision making and actions taken. In general, policies are described as three types, *action*, *goal* and *utility* [54].

- **Action Policy** - determines what action is taken depending on a given *current* state. This type of policy typically takes the form of (IF, THEN) conditions. The *condition* will specify a specific state or a set of possible states. It is presumed that the *design* author, knows what *state* will be reached depending on the actions taken. This is deemed necessary, to ensure the system is exhibiting rational behaviour.

- **Goal Policy** - rather that specifying an exact *current state*, Goal policies rather specify a *desired* state or set of *desired* states. The system responsible for initiating a particular action, can in-turn make a transition from a *current* state to some *desired* state. Rather than relying on a human to create a specific behavioural pattern (i.e. Action policy), the system generates is own rational behaviour itself, using Goal policies. Goal policies allow a greater flexibility, which frees human policy makers from necessarily having to learn about the low-level details of a system function.

- **Utility Function Policies** - this type of policy is a function that expresses the value of each possible state. Utility Function policies provide more fine-grained and flexible specification in terms of behaviour, that you would find in Action and Goal policies. For example, in a situation where multiple Goal policies would *conflict* (i.e. they could not all be achieved simultaneously), Utility policies allow for unambiguous, rational decision making by specifying an appropriate trade-off.

2.2.2.3 3 Tier Model - Inspiration

The Autonomic Nervous System was a topic discussed in [55], in the late 1960s. In this research, observations where being made of that human behavioural characteristics where conspicuously absent from any current computer system. However, human activity can be described as a set of goals and at any time, can be interrupted by environmental changes. Goal priorities can be changed by generating interrupts and therefore, affects intelligent behaviour. Research conducted at the end of the 90s in [3], suggested a 'three tower' model to describe *perception*, *central processing* and *action*. Within this model, there is a provision for 'information' storage and recording of results - (see Fig. 2.3(a)). Further research in [4], introduces the concept of a 3-tier model that has at its lowest level, reactive mechanisms, and at its highest level, reflective processes. Within the reactive layer, external or internal actions are performed immediately. The reflection or meta-management level, provides the ability to monitor, evaluate and control internal processes - (see Fig. 2.3(b)).

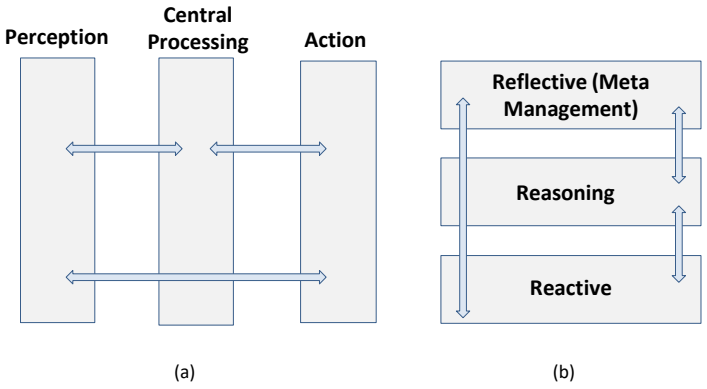


Figure 2.3: Virtual Machine Architectures - 3 Towers (a) and 3 Layers (b) models [3][4].

Research conducted in [5], acknowledges the work carried out in [4], as they also proposed a *human information processing* model operating on three levels. However, although the 'reaction' level is shared by both models, there are fundamental differences between the 'Reasoning' and 'Routine' layers and the 'Reflective Meta Management' and 'Reflection' layers (see Figure 2.4).

The 3-Tier modal (Figure 2.4), described in [5], implies that computer systems

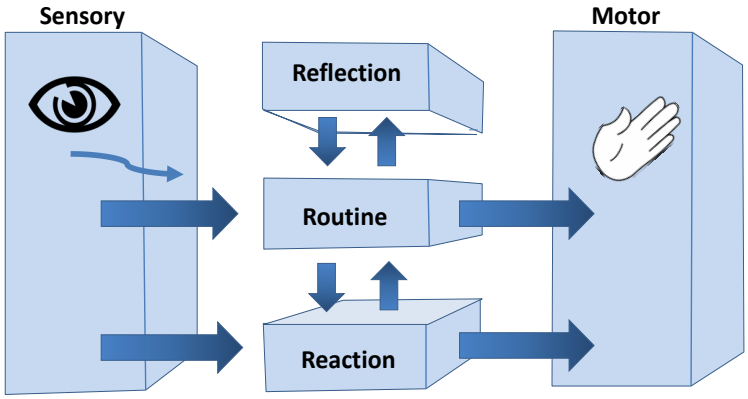


Figure 2.4: Affect and Cognition modal - (three levels of behaviour) [5], adapted from [4].

need to be designed in such a way that they can run autonomously, unattended and maintain high reliability. The three levels proposed are the Reaction level, Routine level and Reflection level. The Reaction level (the lowest level), contains the immediate responses coming from data supplied by sensory systems. The Routine level (mid-level), is complex and involves considerable processing for behavioural selection and guidance. Inputs comes from sensory systems, the Reaction level and the Reflection level. The Routine level can also perform assessments. The Reflection level (highest level), deliberates about itself. It performs operations based on experiences and knowledge. This level has no direct link to 'sensory' or 'motor' systems but relies on input from the lower levels. These 'layers' are explored further in Section 2.2.2.4.

The 3 Tier modal or IMD (Intelligent Machine Design) architecture was then integrated into an 'autonomic computing expression' by research conducted in [5]; in their paper, they state that the IMD modal (integrated with *self* properties at each level*), is more closely related to biological systems and therefore better suited to *certification* as opposed to the MAPE-K model. There is increase in 'intelligence' when moving from the lowest level (Reaction) to the highest level (Reflection). The 'response' speed increases when moving from the highest level (Reflection) to the lowest level (Reaction).

2.2.2.4 IMD and MAPE-K

IMD is significantly different from the MAPE Architecture, both structurally and behaviourally. In this alternative model, behaviours are differentiated in terms of urgency and responding to changes in the environment [6]. The IMD architecture closely relates to how the intelligent biological system works. The IMD architecture proposes three distinct layers (see Figure 2.5) as follows:

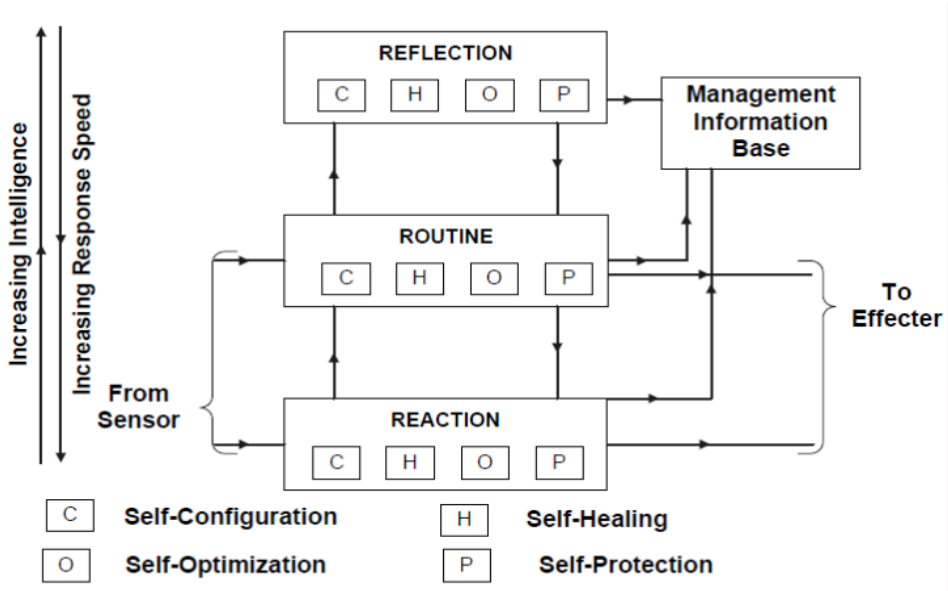


Figure 2.5: An Autonomic Computing expression of the IMD [6].

- **Reaction Layer** - the lower layer, the Reaction layer, is connected to the sensors and effectors. When it receives sensor information, it reacts relatively faster than the other two layers. The main reason for this is that its internal mechanisms are basic, direct and normally hard-wired, therefore, its behaviour is an *autonomic* response to incoming signals. The Reaction layer takes precedence over all other layers and can trigger higher layer processing.

- **Routine Layer** - the Routine layer is more intelligent and skilled compared to the Reaction layer. It is expected to access working memory which contains a number of policy definitions that can be executed based on knowledge and self-awareness. As a result, it is comparatively slower than the Reaction

Layer. The Routine layer activities can be activated or inhibited by the Reflection Layer. If the Routine layer is unable to find a suitable policy for an immediate objective, it hands control over to the Reflection layer. The Reflection layer, the highest level, helps the machine deal with deviations from the norm.

- **Reflection Layer** - the Reflection layer has the responsibility of developing new policies and therefore this layer consumes a larger number of computer resources. The Reflection layer can deal with the abnormal situations, using a combination of learning technologies, specialized algorithms, knowledge databases and *self-awareness*. The Reflection Layer can analyse current data or historic data and identify when to change and selects a policy to decide what to change. The Reflection layer can inhibit or activate processes belonging to the Routine layer through new policy definitions.

All layers in the IMD architecture can implement the four common self-CHOP properties - see Figure 2.5.

2.2.2.5 Autonomy and Autonomicity

Autonomy in the context of computing systems, can achieve its goals without the need for human intervention and therefore can be regarded as self-governing. Examples of autonomy can be seen in space missions conducted by NASA from the mid-1980s on-wards [56]. Autonomicity is regarded as being self-managing. From an autonomic perspective, this would embrace *self-configuring*, *self-optimizing*, *self-healing* and *self-protecting*. Autonomicity differs from Autonomy in that it can adapt to changing environments. Autonomy relies on a pre-set of routines that have been previously designed and cannot be changed. Autonomicity allows for changes to be made when in operation. If a instrument fails then another instrument can be re-configured to take over from the failed component. From a (IMD) perspective (reaction, routine and reflection), autonomicity would be associated with the reaction layer whereas autonomy would correlate with the reflection layer [37]. In Machine Learning techniques, autonomicity is described as levels of automation.

Level 0 is described as having no autonomic characteristics (data is provided by the human developer), where level 4 (the highest autonomicity), is described as being fully autonomic with no human interference [57].

2.2.3 Autonomic Developments

IBM's Architectural Blueprint for Autonomic Computing (2003-2006) [1][47], provided a platform for corporations, researchers and developers to integrate autonomic principles into existing systems or to consider those principles when designing future systems.

In 2003, the Microsoft Corporation instigated its interest in autonomic computing with the Dynamic System Initiative (DSI). Window Server tools such a Resource Management [58], promised to give more control over CPU and memory utilization for managing storage area networks. NASA's interest in Autonomic Computing was evidenced by research conducted in 2005 regarding the ANTS (Autonomous Nano-Technology Swarm) mission [59]. It proposed the use of autonomic properties to organize the behavior of swarms of *pico-class* satellites; self-configuration (loss of communication for one satellite, causes its role to be switched to another local satellite), self-optimization (collecting information on asteroids of particular interest, reducing wasted resources), self-healing (if a satellite is lost due to environmental forces, then it can be replaced by another), self-protection (collisions data reported by individual satellites is shared with the rest of the group, therefore action can be taken to adjust orbits and trajectories). In 2008, research conducted in [60], proposed a method of organizing a Web service-based environment as a collection of agent-based Web services with *self-managing* features. The autonomic manager would be responsible for handling the ACL's (Agent Communication Language) messages from the agents and would guarantee the best web-service for the client.

Autonomic Management has also been investigated by Cloud Computing developers. In 2012, research carried out in [61], proposed that because Cloud systems are large scale and contain multiple distribution centres, they need to be automated and integrated with intelligent strategies for dynamic provisioning of resources in an autonomic manner. In their Cloud system architecture, they introduce

an Autonomic Management system to deal with resource provisioning, application scheduling and security/attack detection. Cloud autonomics is the use of autonomic computing to enable organizations to more effectively harness the power of cloud computing. This is achieved by automating management through business policies. Investigations written in an article in 2015 in [62], envision a future where systems are continuously monitored and if the environment goes out of compliance, the autonomic manager is able to make necessary changes to bring it back in line.

With the ever-expanding scale of the Internet and new devices and technologies being introduced in both wired and wireless environments, network management is a continuing challenge to both industry and the academic world. Research proposed in 2017 in [63], made the case for introducing autonomic system engineering into networks (wired and wireless), and the implementation of self-managing functions and self-adaptability. It introduces an Autonomic Network which contains AN's (Autonomic Nodes). Each AN provides a common set of capabilities across the network called Autonomic Networking Infrastructure (ANI). The main goal is to acquire self-knowledge, discovery, and the information needed for network operations without external configuration. Research in [64], conducted an extensive survey on the possibility of adapting autonomic principles for communication networks. For network security management, autonomic *self-protection* and *self-healing* can be deployed to enforce *denial of service detection and defense*, using the CPN (Cognitive Packet Network) paradigm [64]. There is a comparison between traditional security models (centralized) and autonomic system models (decentralized), where the use of multiple agents can allow for the *autonomic* identification of irregular network activity.

Social Networking has become popular as a means of initiating collaborative work and a method of helping people maintain contacts and communicate up-to-date information. However, it is not in real-time, as it does not rely on event triggers to inform the interested parties. In 2018, research in [65], proposed a Generic Autonomic Social-Collaborative Framework (GASCF) and an Autonomic Adapter (AA), within the Health Care system. This framework can be used to implement

a social-collaborative application that will enable self-managed and adaptive functionality. Multiple AA's will work in cooperation to monitor the systems behavior, track changes and make any necessary changes to hardware and software resources, or send notifications to network nodes. The objectives of this work, in providing an autonomic framework, are similar to some of the research aims outlined in this thesis, where a autonomic generic framework is proposed in Chapter 7.

The future development of Autonomic Computing is very much alive as an article from car manufacturer Ford in 2018 proves [66]. The Ford company is working with Autonomic, a Silicon Valley based company, to build an open cloud-based platform, i.e. Transportation Mobility Cloud (TMC). This platform will be able to manage information flow and basic transactions between a variety of components in the transportation ecosystem (service providers, personalized cars, bicycles, pedestrians, transit systems and city infrastructure including parking and traffic light management). An example of TMC capabilities, is to employ real-time location updates in cities to control traffic flow by dynamically rerouting cars to reduce congestion. A city can ensure that no empty self-driving vehicles are driving on the most important arteries used by people during rush hour. Within the automotive industry the use of customizing is now so well established that popular magazines such as 'CAR' [67], regularly contains reports referring to a vehicle's autonomy level as between 1 and 5.

2.2.4 Organic Computing

Organic Computing [68][69], can be viewed as an extension of the autonomic vision of IBM [1], as it also incorporates self-* management properties. It is a form of biological inspired computing with organic properties. Organic Computing is based on a large collection of autonomous systems, which are equipped with sensors and actuators. Autonomic computing focuses on removing the human users from the system control loop, whereas Organic computing emphasizes the interaction with human users and respecting their needs.

Figure 2.6 shows the Organic System Controller [7]. The SUOC (System under Observation and Control), contains a set of interacting elements and agents

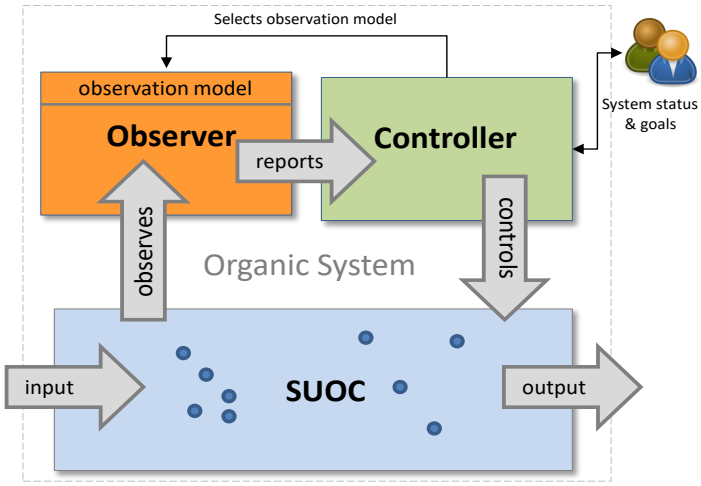


Figure 2.6: Organic Computing - System Controller [7].

which do not depend on the existence of an observer/controller. The Observer will contain pre-processor, aggregator, data analyzer, predictor and log file. The Controller contains mapping, action selector, history evaluation, goals, objectives, simulation model and adaption model.

Organic Computing has been the inspiration for software architectures such as ORCA (Organic Robotic Control Architecture), developed by researchers in [8]. The ORCA architecture is built from Organic Computing Units (OCUs) and Basic Computing Units (BCUs). The characteristics of the ORCA architecture are the scalability and flexibility to property changes of the constituent OCU and BCU units. The BCUs, implement tasks concerned with the robot's control. These tasks can be defined as 'sensor' processing, such as wheel motor movement, sonar sensor control and robot arm control. OCUs on the other hand are responsible for monitoring the correct behavior of the BCUs and to influence their outputs or to change their behavior in the case of malfunctions - see Figure 2.7.

Research presented later in this thesis concentrates on specific aspects of robot control, including both sonar sensors and 'wheel' motor movement. In Section 2.3.4, Organic Computing is explored further, regarding the research that has been conducted in relation to fault handling in mobile robots.

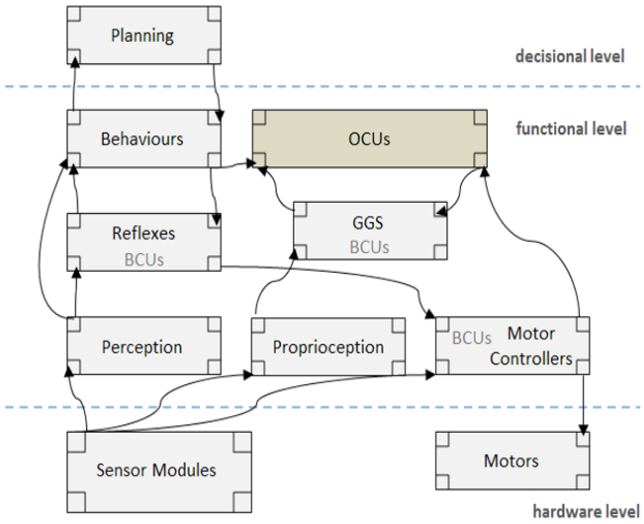


Figure 2.7: Schematic representation of standard ORCA architecture [8].

2.3 Autonomic Fault Handling in Mobile Robots

Mobile Robots are devices that rely on commands that provide instructions for 'motion' and sensor data capture, that report the physical world around them. Mobile robots can either operate in a closed environment such as an industrial factory or hospital; or can operate remotely, such as a pipe inspection vehicle or as a planetary rover. In the case of robots operating in remote regions, it may not be convenient or possible to intervene in order to repair faults. A hardware fault like a damaged wheel on a mobile or planetary robot, can disrupt mission objectives. NASA's JPL Center reported faults on all six wheels from the current Curiosity Rover Mission on Mars [70]. Each of the six rubber wheel casings on the Rover had been punctured by sharp rock material from the planet surface. Consequently, NASA's Mission Control was forced to plan alternate routes for the Curiosity Rover, in order to avoid certain rock types that had caused the damage to the wheels. Autonomic computing implementation can help address the common failure *modes* found in mobile robots. Planetary rovers would obvious recipients for implementation of *self-managing* systems [71].

2.3.1 Fault classification

Mobile robots like all mechanical devices, eventually succumb to some sort of hardware fault or hardware defect. The severity of the fault will dictate the available functionality that the mobile robot can provide. Typical faults for mobile robots are loss of sensors, motorized faults, damaged wheels or power faults. A fault in a system is a variance from the expected behavior of the system [72]. Faults can be classed as follows:

- Permanent (which exists until repaired).
- Transient (which disappear on their own).
- Intermittent (which repeatedly appear).

Component faults don't always show themselves as simply being *non-functional* or *disabled*. In [31] , the authors use *Evidence*, *Fault* and *Value* nodes to identify hardware faults, by recognizing changes in sensor data over time.

In research conducted in [73], robotic failures in relation to NASA missions where investigated. They categorized systems failures as follows:

Wheel Failures - track slippage failures are a frequent occurrence. Self-optimization can be applied to slow down the drive system and minimize further damage.

Control System Failures - [73] stated that these are the most common types of physical failure. If the robot became unresponsive, then recycling the operating system was normally the best course of action to fix this type of issue.

Sensor Failures - the example given in [73], relates to video sensing, where a remote robot could encounter an environment in which dirt particles could inhibit the sensor. Using a self-repair protocols, where a simple sensor wash could re-instate the sensor to full functionality.

Power Failures - when power is low, the autonomic properties engaged protocols that would redirect the robot to the nearest charging station. In the case of a NASA mission, the robot would temporarily power down until the Sun would be in a favourable position to allow a solar panel *re-charge*.

Although investigations carried out in [73], identify key failure areas in mobile robots, there is little to offer in terms of identifying the failure and processing the data available, in-order to formulate a strategy to compensate for the error. In Chapters 4 to 6 in this thesis, we offer case studies on hardware failures in mobile robots. The key strategy for each case study, is Awareness (how is the failure/fault identified?), Analysis (evaluating the extent of the problem through using historical and current data) and Adjustment (Is it possible given the fault data, to establish an compensation strategy to allow the mobile robot to continue to function?).

2.3.2 Fault Tolerance in Autonomic Computing

Traditionally Fault-tolerant systems in robots has become increasingly important especially for robots that operate in remote or hazardous environments. In order to perform tasks without human intervention, mobile robots would require the ability to tolerate and detect internal faults [74]. Many legacy and current systems employ traditional *fault tolerance* approaches. However, with information systems growing in size, many resources are now connected through complicated networks. Systems configurations can change dramatically, even during real-time operations. With information systems now seemingly boundary-less in nature, a single centralized system manager has become less feasible. Autonomic computing offers managed units over a distributed network. Cooperation and collaboration can exist among 'computing' units where there is no requirement for a centralized management mechanism. Autonomic Management systems can also share computing power with different sites, even across enterprises [75].

In Research conducted in [76], implies that dependability and fault tolerance can perhaps be aligned to AC's principle for *self-healing*. However, systems that are incorrectly configured and/or optimized inefficiently, are likely to lead to failures in the future.

Fault Tolerance Control Systems (FTCS), are defined as control systems which can automatically maintain the system stability and maintain an acceptable degree of performance when component failures occur [77]. There are four main FTCS principles:

- **Fault Detection**: to detect that there is something wrong in the system and that a fault has occurred somewhere.

- **Fault Isolation**: to decide which component is faulty and its location within the system.

- **Fault Identification**: to identify the fault and how severely the fault will affect the system.

- **Fault Recovery**: To adapt the system controller structure according to the identified fault. To maintain the system's stability and to continue to operate at an acceptable performance level.

2.3.3 Autonomic Management for fault handling

As part of IBM's original Autonomic concept, *self-diagnosis* and *self-healing* are of particular importance when dealing with system problems. One of the main goals was to create systems that could self-manage and take appropriate action when facing system failure [48]. *Self-diagnosis* is not only concerned with the discovery of potential faults but also the severity and consequences relating to the fault. *Self-healing* is concerned with recovery and repairing itself when dealing with unexpected faults. The main challenge in designing an autonomic system is that all possible fault scenarios cannot be anticipated, it is preferable to design a system that can detect and resolve problems at run-time [78].

Research conducted in [79], investigates that detection of abnormal behavior in a sensor is not always sufficient evidence that the sensor is actually faulty. They proposed to use correlation between sensors as an indication of failure. If one sensor displays faulty behavior due to a faulty component, then is it reasonable to assume the other sensor is not affected by the fault and its reported data still reflects the robot's behavior? However, the same cannot be said about correlated sensors that share the same component, as both sensors can ultimately be affected by the same fault. Research in [78], describes how robotic failure detection, failure recovery and system reconfiguration can be achieved through their Distributed Integrated Affect

Reflection Cognition (DIARC) architecture. Using an ADE (Architecture Development Environment) multi-agent framework, they propose a system that can request information about the current *state* of components within the network. If a failure occurs, for example, in the navigation system, then they can locate a component to take the place of the failed component.

In AC, *self-healing* is the ability to diagnose and re-configure system functions, so that a degree of reliability is maintained. Autonomic Management can be employed to orchestrate dealing with system failures. Giving the Autonomic Manager more decision-making abilities will lessen the need for human intervention. Investigations carried out in [9], proposed the idea of using an AC system to control current AACS (Autonomous Automatic Control Systems). Fault diagnosis and mitigation's are conducted at the *control* layers. They propose that policies are put in place to deal with different types of robotic failure. In Figure 2.8, researchers in [9], create a table of possible fault scenarios within different parts of the robotic system. Depending on the 'cause' of the fault and the 'type' of fault discovered i.e. (1) or (3) vehicle problems, (2) or (4) environmental conditions, they choose a policy that can handle that particular fault situation. Using an Autonomic Management strategy, they apply the following autonomic policies:

Cause	Diagnosis	Mitigation	Policy	Type	Application
Environmental	Wrong path or pose	Pose and motion compensation	If the AACS path or pose is incorrect, the AM actives a compensation mechanism that takes into account the path or pose given as reference	(4)	Mobile Robotics
Mechanical Problem	Module is not responding	Mechanical/Electronic redundancy, alternative capability	If a piece of mechanical hardware or electronic module fails, the AM evaluates the situation in order to activate a redundant system or an alternative capability to replace the faulty one	(3)	Robotics / Automation
Electronic Problem	Device is not working	Quality analysis and improvement	If there is a communication disturbance, the AM analyses the situation in order to active filter and prediction mechanisms to improve signals. If this does not work, it assists the AACS planner to make a decision on what to do next	(2)	Robotics / Automation
Communicatio n Interference	Bad communication	Quality analysis and improvement	If there is a communication disturbance, the AM analyses the situation in order to active filter and prediction mechanisms to improve signals. If this does not work, it assists the AACS planner to make a decision on what to do next	(2)	Robotics / Automation
Evitable Collision	Obstacle detection	Collision-free operation mode	If an obstacle is detected as a potential threat for collision, the AM actives a collision avoidance mechanism that can or cannot involve the AACS planner	(2/4)	Mobile Robotics
Inevitable Collision	Collision with, e.g. vessel, seafloor, other AACS	Collision-recovery operation mode	If the AACS unavoidably collides with something, the AM actives a collision recovery mechanism that takes first the AACS away, and then starts a built-in integrity test	(4)	Mobile Robotics
Operational Problem	Identification, classification, and analysis of the fault	Recovery mechanism	Depending on at what hierarchical level the fault occurs, the AM makes the decision on what is next, or if the AM cannot deal with the problem, it can assist the autonomous controller to deal with it.	(1)	Robotics / Automation

Figure 2.8: Faults Diagnosed and Mitigated at the CONTROL layers [9].

1. **Autonomic Healing** - self-healing is done by defining and isolating possible faults, diagnoses and applying a mitigation plan at the *control* layer.

2. **Autonomic Optimization** - the implementation of self-optimization deals with the efficiency and effectiveness of the AACS, in order to decrease the energy consumption, e.g., applying a strategy that extends battery duration and therefore energy consumption becomes critical in longer missions.

3. **Autonomic Configuration** - the implementation of self-configuration principles, deals with the adaption of AACS in order to improve requirements and goals.

4. **Autonomic Protection** - the implementation of self-protection deals with internal and external AACS network security. Detection, anticipation and identification of attacks to guarantee high security at all times.

The arguments set out in [9], are effective in categorizing how to apply the AC concepts to managing mobile robots and robotic systems. However, this is a high level approach and there is no detail on how to systematically deal with faults in a mobile robot; for example, one of their fault diagnoses **policies** discusses how in discovering a hardware failure, they simply activate a redundant system to replace the faulty one. Research conducted in this thesis, is concerned with analyzing the performance of hardware components within a mobile robot and therefore creating the conditions for anticipating hardware faults, rather than waiting for them to occur. If a fault is discovered, then several policies may be required to analyze and implement a strategy to handle and compensate for the fault. However, research in [9], does make some excellent arguments concerning energy efficiency; as part of the case studies conducted in this thesis, there is in-depth analysis conducted on the effects of lead-acid battery degradation on mobile robot tasks and performance.

In research carried out in [80], their proposal states that in order to handle faults within a network, an implementation of an Autonomic Fault-Management (AFM) control loop is used to assist network personnel. The AFM contained within a network node would be responsible for detecting the presence of a fault, finding the

cause of the fault and finally, removing the root cause. To increase *self-management*, they have formulated a set of criteria in which network personnel should be brought into the loop, if the fault demands it.

- Criterion 1 (The AFM should raise an alarm if incidents reported on the network exceed a predefined threshold).
- Criterion 2 (If the AFM manager is unable to resolve an incident, then this situation should be escalated to network operation personnel).
- Criterion 3 (If an incident is reported but is unknown to the CS (Causality Model) of the network, then Fault-Removal cannot be triggered and therefore should be escalated to network personnel).
- Criterion 4 (The AFM cannot identify the root cause of a fault, then an alarm should be raised to network personnel).
- Criterion 5 (The AFM should raise an alarm if the same type of incidents are repeatedly being reported. This could indicate a failure in clearing down reported faults. If this is the case, then network personnel should be notified).
- Criterion 6 (If the AFM executes a Fault-Removal and it fails, then this should be escalated to the network personnel).
- Criterion 7 (An alarm should be raised by the AFM if a particular component in the network has failed. This should be escalated to the network personnel).

Research conducted in [80], shows that Autonomic Management is not restricted to only detecting/handling faults but also can prove to be a useful tool in reporting to network personnel if the automated systems are not capable of dealing with a fault. This is an interesting concept as Autonomic Management Systems cannot possibly cover all aspects of fault handling. The 'human' presence cannot be entirely removed from the 'loop' and therefore totally automated systems at present, are not achievable.

Resource Management in mobile robots is important, especially for those robots that operate remotely. Poor Resource Management can then lead to the mobile robot developing faults or in a worst-case scenario, complete shutdown. Research proposed in [10], presents a model-based, utility approach to autonomic management of mobile robot resources. Mobile robots can be setup in various configurations. Some configurations may reduce the time a robot has to complete tasks before a battery recharged is required. Other configurations may require that the robot has full connection to a wireless network. The Autonomic Manager is responsible for monitoring the state of the system, analyzing the current state, plan changes required to achieve objectives and to execute these changes. Using various Utility Functions, researchers in [10], attempted to make the best use of bandwidth available to the mobile robot. The size of the bandwidth available, will determine the amount of resources that the mobile robot can access. The autonomic manager will select a particular Utility Function *policy*, which will result in greater bandwidth being made available to the mobile robot.

Table I
RESULTS WITH UTILITY FUNCTION 1

Bandwidth (Mbps)	Sonars	Laser	Video	RGB	Width	FPS
1	on	off	off	–	–	–
2	on	on	off	–	–	–
6	on	on	off	–	–	–
9	on	on	on	off	640	3.00
12	on	on	on	off	640	3.00
18	on	on	on	off	640	3.00
24	on	on	on	on	640	3.00
36	on	on	on	off	1280	3.00
48	on	on	on	off	1280	3.00
54	on	on	on	off	1280	3.00

Table II
RESULTS WITH UTILITY FUNCTION 2

Bandwidth (Mbps)	Sonars	Laser	Video	RGB	Width	FPS
1	on	off	on	off	640	0.42
2	on	on	on	off	640	0.14
6	on	on	on	off	640	1.84
9	on	on	on	off	640	3.12
12	on	on	on	off	640	4.40
18	on	on	on	off	640	6.96
24	on	on	on	on	640	3.17
36	on	on	on	off	1280	3.66
48	on	on	on	off	1280	4.94
54	on	on	on	off	1280	5.58

Figure 2.9: Utility Function policies for controlling bandwidth resource [10].

Figure 2.9 shows how different Utility Function policies allow for different uses of bandwidth. Depending on the task the robot needs to perform, certain components are turned 'off' so that band-width is preserved. If the bandwidth available is low, then the robot is put into an operating mode in which the range of tasks it can execute is significantly reduced.

Research carried out in [10], proposes a Autonomic Management system to handle resource allocation for a mobile robot. This type of resource management is

significant especially if the mobile robot is operating remotely. The research carried out in Chapter 6 of this thesis, focuses on how a mobile robot can complete tasks when dealing with battery degradation. If power resource is low, then some tasks may not be achievable. Reducing the power required for other components within the mobile robot will provide more resources to allow tasks to be completed.

2.3.4 Organic Computing - fault handling in robots

Although work conducted in this thesis is mainly concerned with Autonomic Computing, much can be learned by investigating other fields such as Organic Computing (OC), which relates to this research. Using the architectural model ORCA [8], first discussed in Section 2.2.4, research carried out in [81], shows how a robot system can apply *self-adaptive* techniques even when faced with a major hardware failure. The ORCA architectural model was employed for developing *hexapod* robots. The researchers in [81], experimented in developing a robot that could sense malfunction within its leg support mechanism. In order to make fault detection more adaptive, they introduced RADE (Robot Anomaly Detection Engine). If a servo in one of the legs was showing a 'high' current reading, then RADE could detect the anomaly and report it as a malfunctioning leg. If a malfunction was detected, then the robot was capable of initiating a leg amputation routine to discard the faulty leg: a system re-configuration was performed, which would enable the *hexapod* robot to continue with its mission despite losing a leg. Data was carefully analysed from the robot's individual 'leg' components to establish if there was a possible fault. They then used *re-configuration* algorithms to compensate for the missing limbs. The research in [81], is very relevant to the work carried out in this thesis. One of the main objectives common in all the case studies in this thesis, is to establish possible *compensation* algorithms, that will allow a mobile robot to continue to function even with a faulty component or sensor.

2.4 Generic Autonomic Fault Architectures

Research conducted in this thesis is not only concerned with autonomic fault detection and handling, but also with the question of how common elements found in the

experimental case studies, can be brought together to form a generic autonomic fault architecture. This *generic* architecture is based on the MAPE-K Autonomic Architecture described in [1] and the IMD Architecture in [5] - (see Section 2.2.2.3). This Section of the Literature Review discusses how researchers have applied the Autonomic MAPE-K architecture to their field of work and identifies areas of published work which are applicable to research carried out in this thesis.

2.4.1 Using and Adapting the Autonomic Model: MAPE-K

In remote areas, collaboration between mobile robots is particularly important. Research in [82], has implemented autonomic control loops to coordinate actions between different mobile robots. Using the MAPE-K control loop, knowledge is passed between nodes (robots), such as vision processing and communication protocol. On each robot, the Autonomic Manager needs process data supplied locally and information supplied as (knowledge) from the collective. Researchers in [82] have employed the Autonomic model to great effect by creating a sensor network using corroborative mobile robots. They can now monitor a large area using reduced power capacity but at the same time, increasing performance.

Resources for Cloud computing are made available through networks or through the internet itself. In Cloud Architectures, the *cloud* server will manage and control all applications [34]. Research conducted in [11], shows how the Cloud Controller implements a utility function that involves end-users and providers. The key property of the Controller is robustness. The Controller needs to deal with activities such as erratic workloads, resource congestion, multi-tier applications and scalability. They have adapted the MAPE-K architecture (feedback Loop), as a means of managing their Cloud System - see Figure 2.10.

Monitor workloads provided by applications, **Analyse** the input data as a means to discover possible faults, **Plan** a strategy for initiating an action *policy* for resource management, **Execute** the plan or policy for a specific platform and incorporate any shared **Knowledge**. Research conducted in [11], shows how the MAPE-K architecture can be incorporated into an existing Cloud architecture. This gives the existing architecture a robustness and the ability to use feedback as a

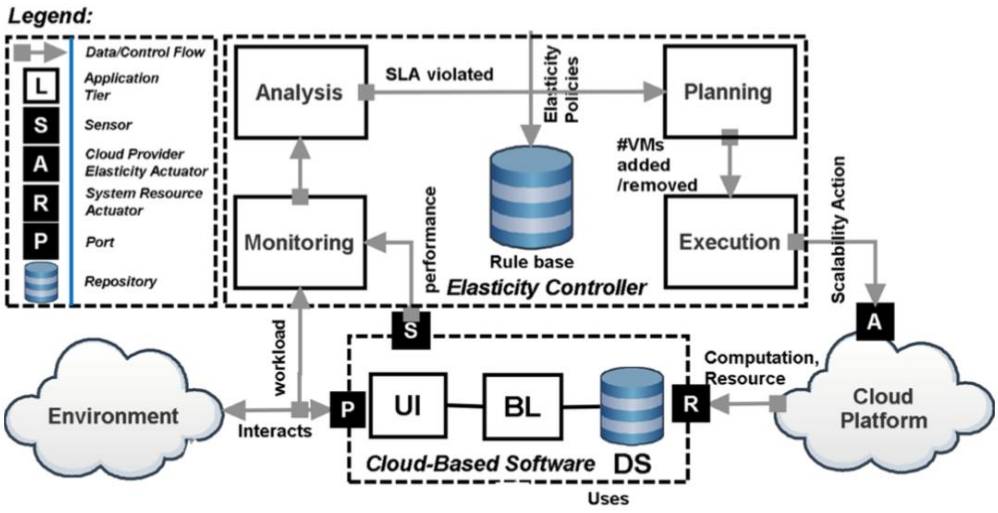


Figure 2.10: MAPE-K Control Loop for the Cloud [11].

means of error correction.

Smart Cites [83], are an example where Wireless Sensor Networks (WSNs) can provide a basic infrastructure for collecting information such as traffic flow, pollution and weather monitoring, Research conducted in [12], shows that WSNs are expected to be *self-managing* systems in order to tackle the increasing complexity of modern-day smart environments. Research in [12], put forward a framework based on the MAPE-K architecture as a means of *self-healing* sensor data supplied online.

1. **Monitoring** - monitoring and analysis as a function of the monitoring process. Monitoring collects data such as topology and configuration properties. The Monitoring process is also responsible for collecting real-time readings from sensors and retrieves historical data from previous sensor readings. The Monitoring process uses a *filter* process to isolate the symptoms. If 'symptoms' are found, then this information is passed to the Analyse stage.

2. **Analyse** - fault detection, diagnosis of faults and finally, classification of faults. Sensor data received from the Monitor process is analyzed to detect faulty readings. Symptoms are process and if changes are required, the fault data is passed to the Plan stage.

3. **Plan** - initiate system recovery and fix any outstanding faults as a function of plan/execute. The Plan process determines an action plan to recover from the fault. The Plan process can either take the form of a complex work-flow or it might be as simple as a single command instruction.

4. **Execute** - alerts the user of changes to the system as a process of plan/execute. The Execute process changes the behavior of the managed resource based on the recommendations supplied by the Plan process.

Sensor data received from the Monitor process are analysed to identify fault readings within the sensor data. In the case of a sensor reading being detected as faulty, a shared Knowledge Base is implemented to classify the fault type. A fault model relating to the classification, is initiated at the Plan stage. During this process, the readings from faulty nodes are corrected. After the fault is diagnosed, the Analyze process will update the Knowledge Base. The aim is to make sure all the sensors across the framework are synchronized and consistent. The Analyse process will then send *self-healing* requests to the Plan process. The Plan process will then decide if appropriate action is needed to either heal the fault or to notify Users to take any further action. Figure 2.11, shows how the researchers in [12] have adapted the MAPE-K model for *self-healing* of sensor data within a WSN.

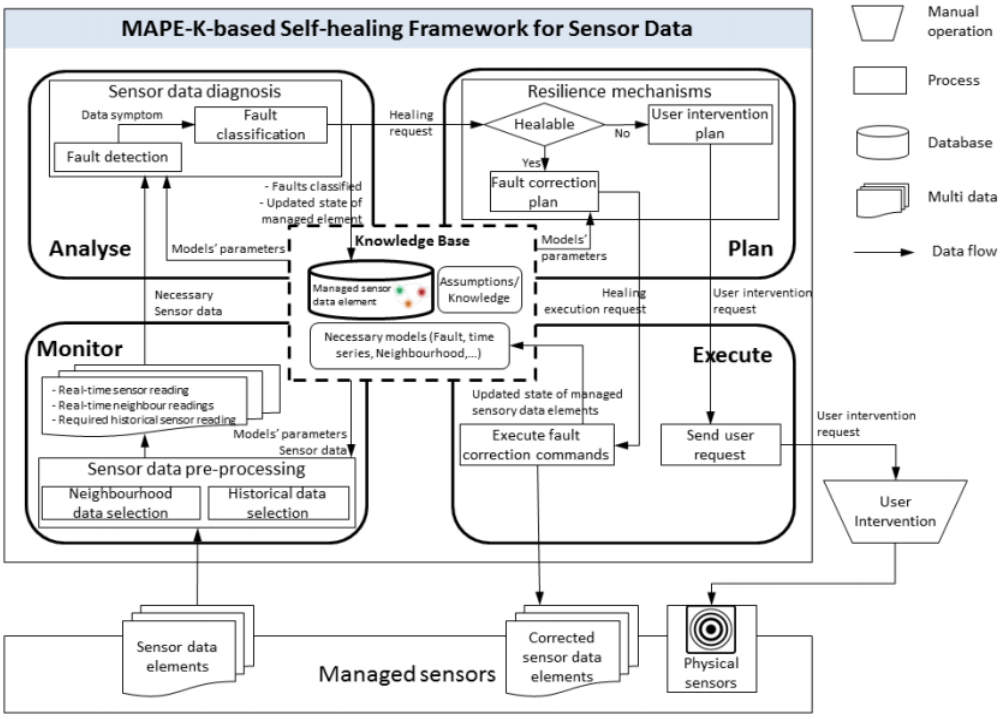


Figure 2.11: The MAPE-K-based framework for *self-healing* of online sensor data [12].

Although research conducted in [12] relates to WSNs, the fundamentals employed for fault detection are also relevant to the work carried out in this thesis. Using the Monitoring process and combined with historical data, the *awareness* of a possible fault can be identified. When a fault is identified, a classification process is used to identify what policy plan can be implemented to handle the fault. The policy is executed in-order to correct the discovered fault. If a fault cannot be healed, then the User is notified to take some action.

Aggregate Computing is a recently proposed framework to build CASs (Collective Adaptive Systems). Research conducted in [13], presents a proposal to bridge the gap between the MAPE autonomic manager and a fully distributed *self-organizing* CASs. It presents an example model of *collective-adaption* that is built around the concept of *ensemble*, which is a collection of autonomous agents which collaborate to perform tasks. Each agent implements a MAPE loop that allows for the interaction with other agents. Figure 2.12, shows how two agents can communicate via their MAPE loops. Each agent is in a *Monitoring* state, while executing

its tasks and monitoring the environment through active handlers. If an issue is triggered between agents (1) or an agent is asked to solve an issue (4), then the *Analyze* state is activated. Based on the triggered issue, the corresponding *Analyzer* is called (2) and (5). The *Planning* state is then activated, where all agents involved in the issue provide a resolution process (Agents A and B). The best resolution is selected (6), and the *Execute* process commits the request as being completed (7-9).

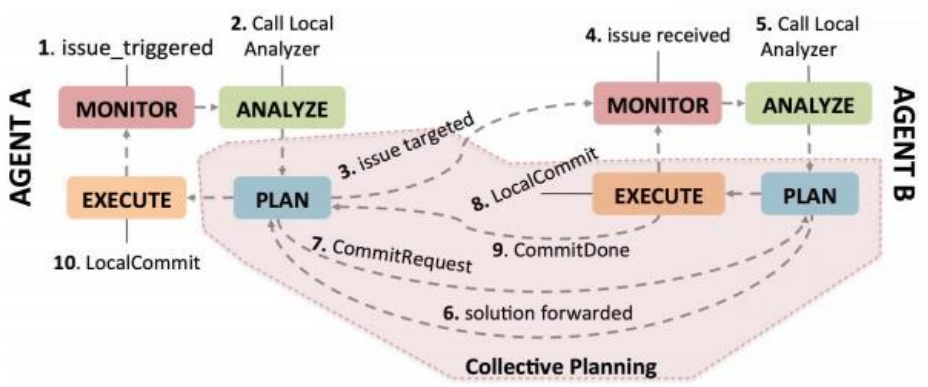


Figure 2.12: Collective Planning using the MAPE architecture [13].

Research in [13], has proposed the possibility of modelling CAS systems as an aggregation of multiple MAPE loops. In future work, they propose to extend their experiments by adopting the MAPE-K model, where the K (Knowledge) component can be used to share knowledge among agents and therefore make more context-aware decisions.

The generic fault autonomic architecture in this thesis, can be applied to the operation of Cognitive Robots. A cognitive robot possesses the abilities to solve problems, plan its own tasks and learn from experience just as intelligent systems in nature do. Research conducted in [14], has put forward an plan for a robust planning framework for cognitive robots. Using the MAPE-K architectural model, their framework contains six modules - Planning module, Scene/Object module, Execution module, Monitoring module, Reasoning module and Learning module. These modules are connected to sensor and motor interfaces of a robot system - see Figure 2.13.

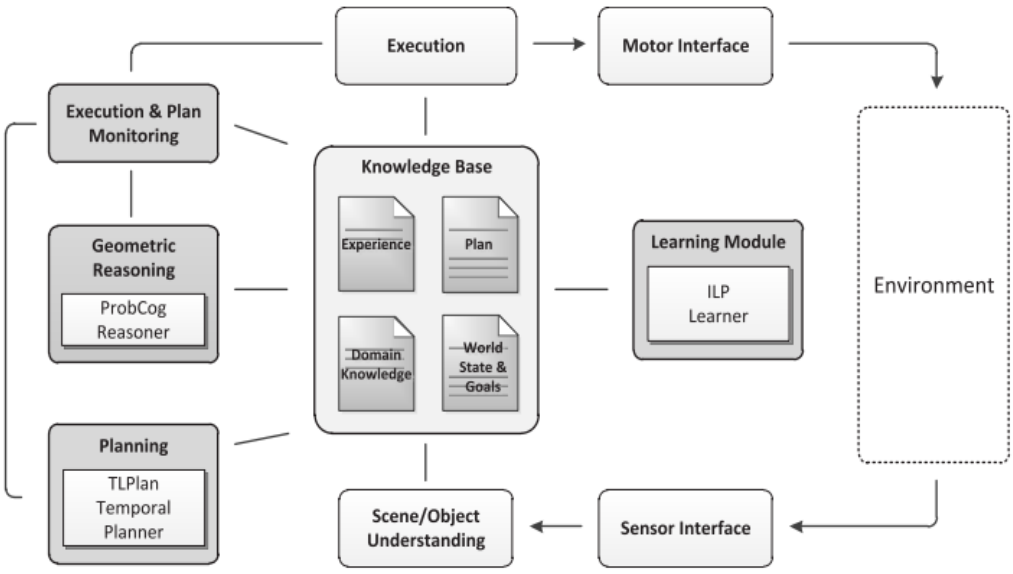


Figure 2.13: Robust Planning Framework for Cognitive Robots [14].

The Planning module constructs high-level plans for given tasks. It uses the Knowledge Base to aid in the generation of valid plans. If a plan for a given state is found, then the Execution module can execute the actions within the plan. The Scene/Object module is responsible for updating the Knowledge Base, based on data gathered from the environment. The Plan and Monitoring modules receive information from the Execution module, which monitors the execution of the plan and if appropriate, detects any failures. The Reasoning module reasons about any failures that are discovered and alters the parameters of the failed action if required. The Learning module is responsible for the robot to adapt itself based on experience gained through actions and how the environment may be affected by those actions.

The research conducted in [14], shows that it is not always necessary to treat the MAPE feedback back loop as one step after the next. After analyses, it may be required to return back to the monitoring process if circumstances have changed within the environment. Updating the Knowledge Base may result in parameter changes and thus tolerance values to detect faults can also change. Tolerance Adaption is an important part of the generic architecture proposed in this thesis. Tolerance values need to be dynamic so that faults are not reported unnecessarily.

2.5 Summary

This Literature review set out to introduce the concept of Autonomic Computing and how AC has developed over the years since its conception by IBM in 2001 [41]. The Autonomic Architecture (MAPE-K) has been reviewed as well as alternative architectures such as Intelligent Machine Design (IMD) and Organic Computing.

Fault classification plays a key role in how faults are interpreted. By their very complex nature, mobile robots will likely suffer some type of hardware failure during their lifetime. With mobile robots that operate remotely, faults become particularly challenging, as these robots are not easily retrieved for repair (for example, if the vehicle is a planetary rover), and the fault may affect how the robot performs its tasks. The Autonomic Manager can provide a method for *self-diagnosis* and *self-healing*. Using the autonomic MAPE-K feedback loop as a basis, many researchers have developed various methods for detecting and handling fault scenarios. However, there is a noticeable lack of the employment of 'early detection' methods. In most research cases, there is a reaction to the fault, rather than a prediction that a fault may occur. The use of the MAPE-K Knowledge Base is fundamental in using historical and current data to establish if a fault has occurred or if a fault may be pending.

Section 2.4.1 in this Literature review, examined how researchers have extended the MAPE-K architecture and integrated into their existing architectures. A generic autonomic architecture can be implemented to handle general system activities but also handle various types of violations and faults. A common theme proposed by researchers are the use of policies to handle the various fault scenarios. Detecting the fault is only part of the problem; finding a policy to compensate is a key part, as this can restore the system to its original state or at least to a *functioning* state. However, policies are only useful if the type of fault occurring is predictable. Not all fault scenarios are possible to predict; therefore, policies need to be dynamic so that they can process data that is produced by *unpredictable* faults.

One of the main objectives of this thesis, is to investigate and experiment with faults scenarios within a mobile robot system through case studies. Common ob-

servations and solution elements are then drawn from each case study and used to design a generic autonomic fault architecture. The motivation here is to not only to detect faults and provide compensation policies but also to highlight that designers of mobile robots should endeavour to allow more programmatic access to components within the robot, so that if a fault occurs, *compensation* strategies can be more effective.

Chapter 3

Research Hypothesis and Method

This chapter presents the overall research directions of this work through guiding Research Hypothesis. It also presents the method used to shed light on the guiding research hypothesis.

3.1 Goals

1. Research the main attributes associated with the MAPE-K and IMD models and establish if they do enough within the autonomic robotic community or is there areas that can be improved upon.

2. Identify *self-awareness* (i.e. becoming aware), when establishing a fault scenario within a mobile robot.

3. Using case study methods, identify common design patterns in-order to develop an autonomic generic architecture for handling component faults in mobile robots.

3.1.1 Design Patterns

What design patterns can be identified when defining a solution for fault detection, analysis and compensation in mobile robots? A *design pattern* is a reusable solution to a recurring problem [84]. Re-usability: reusing existing knowledge from previous designs can reduce development time and decrease complexity. Software patterns can be applied at three levels: analysis level, design level and implementation level [85]. This type of design pattern configuration can be applied within the

Methodology.

3.2 Research Method

3.2.1 Case Study Methodology

A case study methodology allows an investigation to be carried out relating to a *event* within its real-life context. In this thesis, case study attributes are as follows:

Resources - are provided by a fully working robot (Pioneer P3-DX), which is fitted with multiple sensors. The robot has an internal computer board which can be assessed remotely in order to run tasks.

Units of Analysis - data from multiple tasks is used to establish if 'threshold' values are exceeded. Using multiple tasks ensures that evidence gathered reflects the behaviour of components and sensors accurately.

Generalization and Reliability - Through experimentation, establish that sensors are reporting correct data. Compare the sensor results with established parameters, so that faults identified are legitimate. Validate 'fault' data with actual observations.

Internal Validity - establish patterns within and between each case study. Are there distinct relationships in the results produced from each case study? It is important when developing a generic architecture to find common elements. The final generic architecture, while performing its goals, must remain as simplistic as possible.

A successful case study demonstrates the potential of the generic architecture in providing evidence of how faults in robots can be identified, analysed and if possible repaired. In this thesis, three case studies will be used assist the development of an autonomic 'generic architecture'; each case study provides a component *fault* scenario within a mobile robot. To validate the 'generic architecture', a further case study will be implemented. The case studies will form part of a 3 phased structure - see Figure 3.1.

Creation Phase - the outline proposal of taking elements from the MAPE-K and IMD models to create a new generic architecture.

Evolutionary Phase - as each case study (1-3) is investigated, results from using *awareness*, *analysis* and *adjustment* methods will contribute to developing the autonomic generic architecture.

Validation Phase - will demonstrate how the fully formed autonomic generic architecture can be applied to a further case study (4).

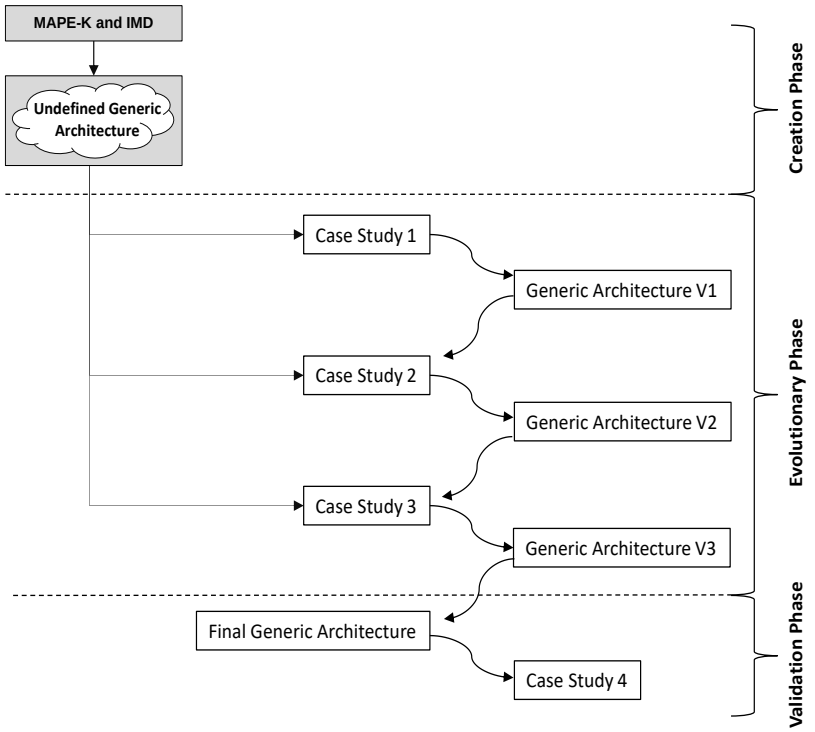


Figure 3.1: The proposed autonomic generic architecture phase development using a case study approach.

3.2.2 Generic Architecture (awareness)

In robotic research, the notion of *self-awareness* is still a major topic of discussion. Can a machine develop self-awareness and by doing so, understand its environment, be in control of its own actions and be responsible for those actions [86]? In research carried out in [87], a *self-simulator* was created that allowed a robot arm to *self-model* itself by collecting multiple trajectories, each comprising of one-hundred *pointers*. The researchers in [87], deliberately created a 3D deformed part of the arm and the robot was able to detect the 'change' and re-train its *self-model*.

Self-awareness forms a fundamental part of the 'generic architecture' re-

searched in this thesis. In Autonomic computing, *self-aware* is a system's ability to 'know itself'. A *self-aware* system must have knowledge of internal components, their current status and their historical journey.

3.2.3 Generic Autonomic Fault Architecture (Creation Phase)

For the purposes of this thesis, the Generic Autonomic Fault Architecture will be referenced as AIFH (Autonomic Intelligent Fault Handling) architecture.

The AIFH (Autonomic Intelligent Fault Handling) architecture developed for this research takes elements from the MAPE-K and IMD architectures. The MAPE-K architecture can be modified so that subsets of *monitor*, *analyse*, *plan* and *execute* functions can be utilized [1]. The AIFH architecture can also adapt attributes from the IMD architecture. The IMD architecture uses a layer design but only the Routine layer and Reaction can communicate with the sensors and effectors. Knowledge is only accessible through the Reflection layer [5] - see Fig. 3.2.

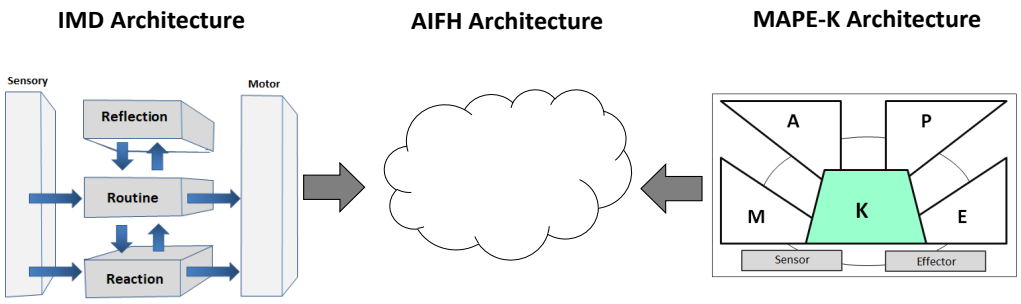


Figure 3.2: AIFH Architecture contains attributes from both the MAPE-K [1] and IMD models [5]

In Section 3.1, a research question was proposed "Does the MAPE-K and IMD models do enough to guide the development of autonomic systems for robots or can they be improved upon?". The MAPE-K architecture follows a series of steps - *monitor*, *analyze*, *plan* and *execute* (using a feedback loop), which could form the basis of a fault-management architecture. However, does simply *monitoring* events and declaring an incident fulfil the demands for an *autonomic* system? The IMD architecture uses 'intelligence' in its decision making in terms of *reflection*. Therefore, to attain the desired *autonomic* performance, *intelligent monitoring* would be

preferable. Not only are *reactive* incidents reported but also *trends* in component behavior is also reported. If a component or sensor can be identified as a possible risk to the robot's operational status, then this could prevent a system failure in the field. Therefore, in this thesis, *intelligent monitoring* will aid in answering the question, *'Becoming 'aware', what does it mean and how does it work?'*. In other words, not simply reporting a fault but making intelligent decisions about impending faults.

3.2.4 SDLC Methodology

To design the generic architecture, the SDLC (Software Development Life Cycle) methodology will be applied within each of the case studies (see Chapters 4, 5 and 6). SDLC promotes a well-structured approach to defining each stage of the case study. SDLC is a standard process of the methodology to structure all steps necessary to analyse, design, implement and test the system. In general, SDLC methodology contains the following steps:

1. The system requirements are defined. In particular, what goals are expected and what resources are needed to achieve those goals.

2. The proposed system is designed. The plans are laid out for the architectural design and what operating systems, software programming and reporting mechanisms are required. Design patterns identified.

3. The system is implemented. Analysis performed on collected data by developing specialized Algorithms.

4. The system is tested. The analysed data is evaluated to produce reports and specialized algorithms are developed that can alter system parameters if required.

5. The system is evaluated. Test results should be evaluated for consistency. Have the goals laid out in the requirements been achieved? What lessons where learned during the SDLC process?

The main objective in each case study is to develop the AIFH basic model (see Fig. 3.2), into an *autonomic* generic architecture that can be used to handle

component faults in a mobile robot. The case studies are specific to certain systems within a mobile robot. The first case study is centred around the *drive system* of the robot and how a wheel fault can affect navigation objectives. The second case study is centred around *object detection* sensors; this case study explores how faults in sonar sensors can limit the ability of the robot to detect objects. The third case study is centred around the *power resource* within the mobile robot; this case study examines how degradation in the battery power supply can affect how the robot performs in completing its tasks. Using the SDLC Methodology, each case study will be laid out as follows:

Requirements: - sets outs the research objectives for the case study and what goals are to be achieved. The fault scenario (experiment), is explained and what systems in the robot will be affected. Laboratory setup is explained and what parameters are used to carry out the experiment. Resources required for the experiment are identified, i.e. hardware and software requirements.

Design: - will show the outline design of the generic architecture and how it can be applied to evaluating sensor data from a mobile robot. How the evolution of the AIFH model can identify *design patterns* and layers within the architecture can be identified.

Implementation: - shows how data from the sensor/effector/power system data can be monitored and therefore implement *self-awareness*. The development of specialized algorithms to process the data and evaluate the extent of the fault.

Testing/Evaluation: - When the fault data has been analysed, test strategies can be implemented to establish a method of *compensation* for the fault. Evaluating the results will establish if the case study objectives have been achieved.

3.3 Summary

In this chapter, the *Goals* of the thesis research where identified. The *Research Method* chosen was based on case study Methodology. The case study *phases* where identified so that the *autonomic* generic architecture can be developed as each case study is completed. The basic generic architecture was designed based on the

MAPE-K and IMD architectural models. Finally, SDLC Methodology is proposed as means to develop each case study. SDLC offers a well-structured approach, as it defines each stage of the case study. Chapters 4 - 6 will contain a unique case study for a fault scenario within a mobile robot. These case studies will help to contribute to the final *Autonomic Generic Architectural* design explored in detail in Chapter 7.

Chapter 4

Self-Adaptive Mobile Robot Wheel Alignment - Case Study

4.1 Introduction

Physical failures in mobile robots can affect hardware components such as effectors, sensors, power units and communication. Section 2.3 discussed Autonomic Fault Handling and showed how *self-diagnosis* was concerned with discovery and evaluation of a fault and *self-healing* was concerned with recovery and repair from a fault. In this Chapter, a case study is presented which involves the processing of sensor data and how that data is interpreted as an indication of a hardware fault. The case study *research example*, shows how a fault in the wheel component of a mobile robot can affect how the robot performs regarding *dead reckoning*. Using the AIFH autonomic model as a base (see Section 3.2.3), the architecture is expanded to handle hardware fault diagnosis in mobile robots. Processes are implemented to establish if a fault exists (Awareness), evaluate the extent of the fault (Analysis) and finally, to compensate for the fault (Adjustment). These processes are put in-place so that the mobile robot can still perform at an operational level.

Wheel Fault detection in mobile robots has been the subject of some investigation. Research conducted in [88], shows how wheel faults on mobile robots can be detected using LMN's Local Model Networks. Comparisons are made between the actual wheel parameter output with the LMN *model* output. If the values are

significantly different, then the presence of a wheel fault is greatly increased. The use of a Kalman Filter for detecting wheel faults is explored by researchers in [89]. They can detect when a mobile robot is experiencing *wheel slippage*, due to contact with an object in the local environment. Research conducted in [90], explores the use of mathematical models to generate *residuals*. The mathematical model of a system process is run in parallel to the real system generated by the robot itself. The difference between the measured process variable and the variable estimated through the mathematical model, provides the residual values; the goal is to detect faults as early as possible and provide the User with an adequate warning. Further research conducted in [91], explores the use of multiple robots that implement a common shared *co-operation algorithm* to track the trajectory of individual robots. The trajectory of the robots is verified using beacons placed within the area in which the robots operate. If the trajectory of a robot does not follow the *desired path*, then a fault is declared against the offending robot. However, their research has yet to develop a compensation strategy for the estimated fault.

Although there has been considerable research in the detection of wheel faults on mobile robots, there is very little research concerning the application of the Autonomic Model to detect wheel faults. There is also a lack of research conducted in the form of compensating for wheel damage in a mobile robot when a fault is detected. For the case study presented in this Chapter, wheel faults are introduced into real robots and handled using the Autonomic Management approach.

This case study is organized as follows: Section 4.1.2 describes how the Research Methods is implemented by utilizing the SDLC (Software Development Life Cycle) model. Section 4.2 presents Conceptual Requirements which describe the Research question, goals and resources needed to complete the case study. Section 4.3 presents the Conceptual Design and describes how the Autonomic Management System has been adapted to provide *self-awareness*, *self-analysis* and *self-adjustment*, when dealing with mobile robot faults. Section 4.4 presents Implementation of the wheel alignment fault scenarios and how Awareness, Analysis and Adjustment have been used to evaluate and compensate for a fault. Section 4.5

demonstrates through testing, how the *compensation* algorithm for the wheel alignment fault performs; results from the testing are then analyzed. Section 4.6 presents an evaluation of the case study. Section 4.7 concludes the case study with a summary statement.

4.1.1 Introducing the basic AIFH model

The AIFH architecture consists of two sections: the System Manager and the Autonomic Manager. The System Manager provides the communications between the hardware components (i.e. sensors and effectors), to the Autonomic Manager. The System Manager also provides the interface between the robotic system and the human interface. The Autonomic Manager contains 3 main elements: Awareness, Analysis and Adjustment. Each of these elements are connected via a feedback loop which orchestrates the flow of data within the Autonomic Manager. As each case study is developed, the AIFH architecture will evolve, as different fault patterns are identified and applied to the final generic architecture.

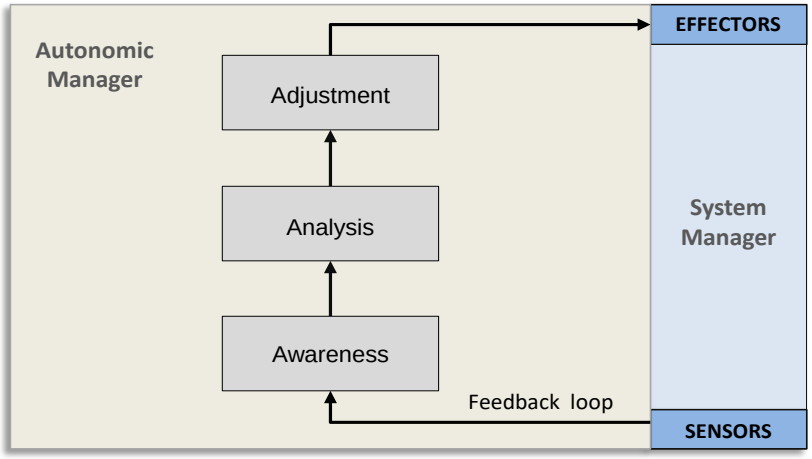


Figure 4.1: The basic AIFH model.

4.1.2 Research Method

The SDLC model approaches solving a given problem in well-defined steps. Figure 4.2 shows how the SDLC Model can be applied to the Autonomic Self-Adaptive Robot Wheel Alignment case study.

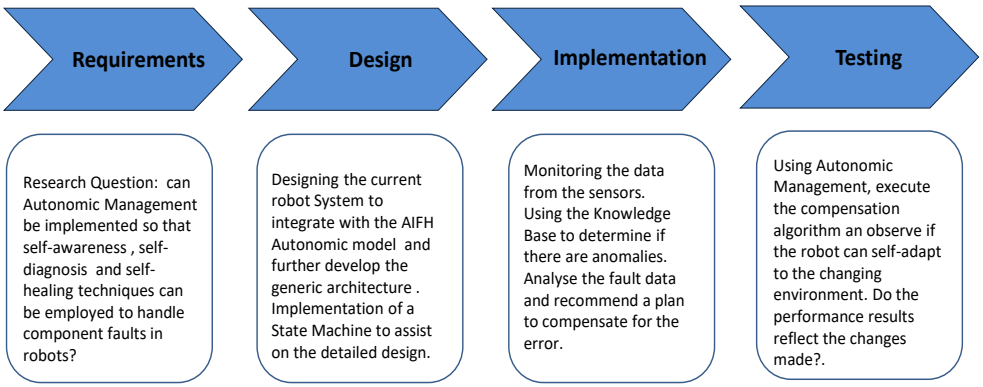


Figure 4.2: SDLC Model used in the research methodology for Autonomic Wheel Alignment

For the purpose of this case study, the SDLC model from Figure 4.2 is employed. SDLC model ensures that all the work carried out within the case study is documented and results generated.

- **Requirements** - lays out the broad research objectives of the particular case study. It is a detailed investigation of the system and is carried out in accordance to the objectives proposed. It involves a detailed study of various operations performed by the system and their relationships within and outside the system [92]. To develop the research question successfully, physical components such as the Pioneer P3-DX robot and LMS 200 laser are required. The robot will also require wheels that are in optimal condition and wheels that are damaged (damaged wheels are achieved by altering the internal foam structure of the robot wheel, thereby causing the effect of a 'flat' tyre).

- **Design** - based on the information collected at the requirements stage. The logical system design is arrived at as a result of system analysis and how it is converted into physical system design. The SDLC process moves from the **what** in the requirements phase to the **how** in the design phase [92]. For this case study, the design is concerned with applying the Autonomic Model to handle how the robotic system can adapt to the discovery of a component fault. In Chapter 3, Section 3.2.3, the initial concept of the AIFH architecture

was introduced. In this case study, the findings accumulated in the *autonomic wheel alignment* research, will provide a means to develop the AIFH architectural model. The layers contained in the AIFH architecture are *Awareness*, *Analysis* and *Adjustment*. *Self-Awareness* is the most important, as it is an indicator to the mobile robot autonomic manager that there is a fault within the system. *Self-Analysis* is used to evaluate the extent of the fault. *Self-Adjustment* takes the analyzed data and applies a specialized algorithm to compensate for the fault.

The design phase is also concerned with how a State Machine design, which is based on the MRDS (Microsoft Robotics Development Studio) CCR (Concurrency Co-ordination Runtime) [58], can be implemented to handle the complicated process of fault analyses and fault compensation on the mobile robot.

- **Implementation** - the system design needs to be implemented to make it a workable system. To create the wheel alignment experiment, the robot is first fitted with two wheels working at optimal performance. The robot tasks are performed, and the wheel alignment data is collected. The second phase of the experiment involves fitting the robot with a damaged wheel. The robot tasks are then run again, and the wheel alignment data is collected. In this case study, programming is implemented to allow the Autonomic Manager access to the sensor data in the AIFH Knowledge Base. Further programming is implemented to evaluate any fault data that supplied by the *Monitoring* process. When the fault has been identified, then additional programming is required for formulate a possible compensation strategy.

- **Testing/Evaluation** - involves system integration and system testing of the programs and procedures coded at the implementation phase. Testing is a method of testing the system against the requirements and design. For this case study, testing will involve evaluating the P3-DX robot on how wheel alignment performs with 'no wheel damage' and 'with wheel damage'. Test

Strategy:

1. Run the robot tasks with the robot fitted with two fully operational wheels.
2. Does the robot arrive at the expected destination when operating with no wheel damage?
3. Is the wheel alignment data for the 'no wheel damage' test, within tolerance values?
4. Does wheel alignment data suggest that there could be a possible wheel fault in the future?
5. Run the robot tasks with the robot fitted with one damaged wheel.
6. Does the robot arrive at the expected destination when fitted with a damaged wheel?
7. How much is the alignment *tolerance* value compromised with the robot fitted with a damaged wheel?
8. With the wheel alignment fault identified, run the robot tasks using the compensation algorithm.
9. Is the wheel alignment data for the 'wheel damage' test, within tolerance values?
10. How did the *compensation* algorithm perform when adapting *intervals* during the robot tasks?

4.2 Conceptual Requirements

The Requirements phase in a SDLC model is the most crucial step in creating a successful case study. Requirements define the problem, objectives and the resources needed to complete the study.

4.2.1 Research Question

4.2.1.1 Goals

Using AIFH model (see Fig. 3.2), as a baseline, the AIFH architecture is further developed in this case study to establish if an autonomic architecture can be used to detect and compensate for robot component faults.

- **Awareness** - can *Monitoring* past and present experimental data, allow the Autonomic Manager the ability to decide if there is a wheel fault on the robot?
- **Awareness** - can *Monitoring* past and present experimental data highlight any trends that the wheel alignment data may be suggesting an impending fault?
- **Analysis** - can *Analysis* provide the means to establish the extent of the fault?
- **Adjustment** - can *Planning/Execute* provide a policy that will compensate for the fault in the wheel component?

4.2.1.2 The Experiment

Using the Pioneer P3-DX robot fitted with the LMS 200 laser. The LMS 200 laser is used to measure the distance from the P3-DX robot to the laboratory wall. The laser is important in this experiment as it can accurately measure distance values and therefore wheel *alignment* values can be calculated.

In laboratory conditions, the robot is driven on a parallel course at a fixed distance from the wall for not more than 2 meters. This robot is then turned 180 ° and the procedure is then repeated several times. Before the robot begins each 'run', the laser is used to record the distance the robot is from the laboratory wall. This is repeated when the robot comes to a stop and before it is rotated for the next run. The *distance* data collected by the laser component, is stored on a database for analysis. The first experiment is conducted with both wheels on the robot are fully operational - data is then collected and 'wheel alignment' performance is then evaluated. The second experiment is conducted with the robot fitted with one damaged wheel. Fitting the mobile robot with one damaged wheel is significant, as it shows how a

wheel fault will affect wheel alignment performance. The third experiment is conducted using a slightly damaged wheel. This experiment is concerned with flagging to the User or Mission Control, of a possible impending fault. Although the wheel alignment data will be within tolerance values, the aim of the experiment is to show that even slight changes to wheel alignment readings could mean a possible wheel fault in future operations.

4.2.2 Resources required

4.2.2.1 Hardware setup

Experiments carried out in this case study are conducted using a Pioneer P3-DX robot. The Pioneer P3-DX robot has two independent drive wheels, plus an additional caster wheel for stability. The internal drive uses a Proportional Integrated Derivative (PID) system with wheel encoder feedback to adjust a pulse-width modulation (PWD) at the internal motor *drivers* to control the power to the motors [93] - see Figure 4.3.

Figure 4.3: Pioneer P3-DX research mobile robot.

The P3-DX mobile robot is fitted with an LMS 200 Laser. The LMS 200 is a laser that is capable of measuring distances out to 80m over a 180° arc. The sensor operates by shinning a laser via a rotating mirror. As the mirror spins, the laser scans 180° , effectively creating a *fan* of laser light. Any object that breaks this *fan*, reflects laser light back to the sensor. Distances are calculated based on how long the laser light takes to bounce back to the sensor - see Figure 4.4.

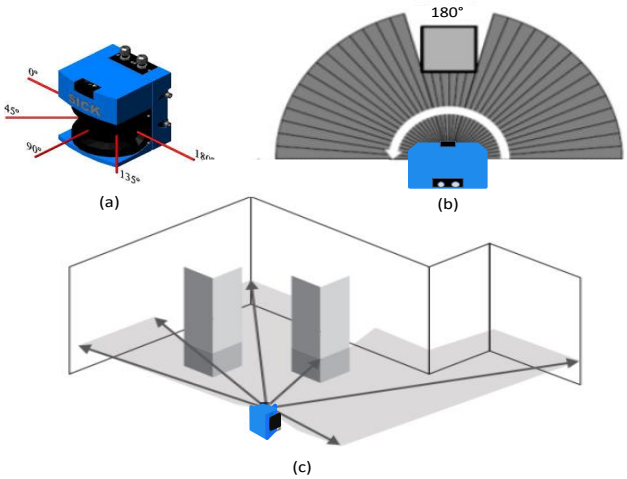


Figure 4.4: (a) The LMS 200 Laser has 180 ° field of view. (b) The laser creates a fan of laser light that scans from right to left. (c) Objects are detected by breaking the laser fan projection.

4.2.2.2 Laboratory setup

For the laboratory experiment (as described in 4.2.1.2), the P3-DX robot is setup parallel to the laboratory wall. The laboratory wall is used as a reference marker. The LMS laser fitted on the P3-DX robot is used to measure the distance the robot is from the laboratory wall. In Figure 4.5, points A1 and A2 represent the distance (using the laser), the robot begins a task from the wall. Points B1 and B2 represent the distance the robot is from the wall at the end of the task. The *tolerance* value is set and stored in a database for reference when tasks are executed. If the B2 value exceeds the tolerance limit, then this would represent a wheel alignment fault.

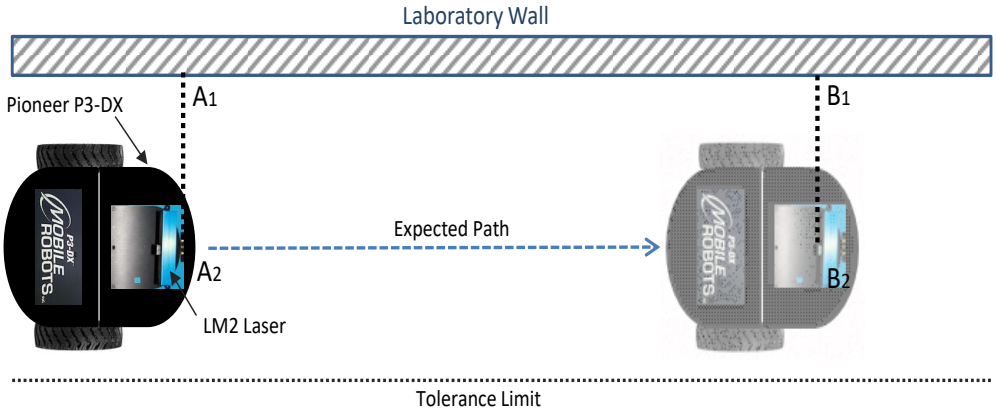


Figure 4.5: Pioneer P3-DX wheel alignment laboratory setup.

4.3 Conceptual Design

The Requirements in Section 4.2 have provided a mechanism to fulfil the case study experiment. This Section is concerned with conceptual design and how the autonomic principles can be applied to dealing with 'wheel alignment faults'.

4.3.1 Developing the AIFH Architecture for wheel alignment fault handling

Using the basic AIFH architecture introduced in Figure 4.1, the Autonomic Manager and System Manager are expanded to form the elements required to process faults in mobile robots. The three processes discussed in Section 4.1.1, Awareness, Analysis and Adjustment, are arranged in 3 layers. The Autonomic Manager manages the communication between each layer and how the *knowledge base* is shared. In the MAPE-K architecture [1], the Autonomic Manager implements an intelligent control loop that is made up of four parts. Each part communicates and collaborates with one another and share appropriate data (knowledge). For the AIFH (Autonomic Intelligent Fault Handling) model, two separate control loops are required - Reactive Loop and Proactive Loop.

- **Reactive Loop** - this control loop is concerned with making decisions based on the current component state. The Reactive Loop is initiated by the System Manager, as tasks are been carried out by the robot. The Reactive Loop passes through each Layer within the Fault Handling Architecture. This control loop is responsible for passing fault data between each layer. Finally, the Reactive Loop will communicate and provide data to the System Manager, if task adjustments are required to compensate for a fault.

- **Proactive Loop** - this control loop is concerned with processing historical data with current data. The Proactive Loop can make decisions based on performance trends from sensors and effectors. This control loop is based in the *Awareness* Layer and reports unusual readings to the User Interface which is based in the System Manager.

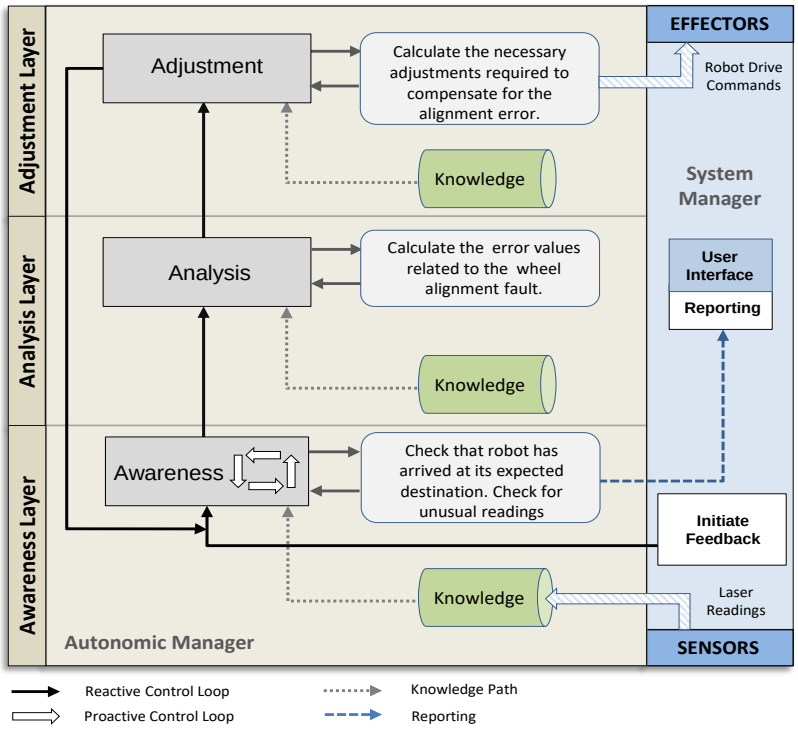


Figure 4.6: Autonomic Management System for Wheel Alignment case study.

Figure 4.6 shows the Autonomic Manager and System Manager for the Wheel Alignment case study. The System Manager is responsible for handling *task* parameters, *sensor* control and *effector* control. While the System Manager is running a task, the *distance* readings taken by the LMS 200 laser (from the mobile robot to the laboratory wall), are recorded into a database as *Knowledge*. As each task is completed, the Autonomic Manager *Reactive* control loop is executed. The *distance* data contained in the database table provides the Autonomic Manager with 'knowledge' of how the robot is performing regarding its wheel alignment accuracy. The *Awareness* Layer is responsible for checking the *distance* readings from each task against a known tolerance value (stored in the database and calculated when setting up the laboratory experiment - see Section 4.2.2.2). If the robot *alignment* data is above expected tolerance value, then the task data is passed to the Analysis Layer. The *Proactive* control loop that is initiated in the Awareness Layer, will process *distance* readings that are within tolerances but are showing a trend that may suggest that a fault may be impending. These 'suspect' readings are flagged to the System

Manager User Interface. The Analysis Layer is responsible for evaluating fault data sent by the Awareness Layer. The Analysis Layer uses policies contained in the *knowledge base* to establish the how much the robot has strayed from its expected destination; this is known as the *error offset angle* value. The data gathered in the Analysis Layer is then passed to the Adjustment Layer. In the Adjustment Layer, the *angle of turn* value is then calculated. The *angle of turn* is important as it provides the mobile robot a *rotation angle* value that will allow the robot to slew back towards its intended path. The Adjustment Layer uses a *wheel alignment compensation* policy from the *knowledge base*, which implements the *angle of turn* value. This policy (algorithm), adapts a stop-rotate strategy to allow the robot to slew back towards its intended path. The System Manager uses the fault *compensation* data from the Adjustment Layer to control the robot's movement via the *effectors*.

The Autonomic Management System for the Wheel Alignment case study shows how autonomic *self-awareness*, *self-analysis* and *self-adjustment* is integral in establishing if there is a fault in the wheel alignment of the mobile robot. It is important that the robot is *Aware* that there could be a possible fault and that it requires investigation. *Self-awareness* is also key, as it not only reports faults that are obvious (like a severely damaged wheel) but it also reports any subtle changes in the wheel alignment data that could predict a possible fault in the future. With a wheel alignment fault identified, then the autonomic *self-adjustment* process is important for the robot to continue to function. When the fault data has been analysed, then this data can then be applied to a *compensation* policy. The *compensation* policy is dynamically designed so that various faults scenarios can be handled depending on the severity of the wheel damage. When the *compensation* policy is applied to the wheel alignment fault, then the System Manager will initiate the autonomic *reactive* control feedback loop and therefore will trigger the Awareness Layer to check that the robot is within expected tolerance values.

4.3.2 State Machine

Software development for this case study was carried out using the MRDS framework. MRDS is a service-oriented programming model that allows the creation of

asynchronous and state-driven applications [94]. Code implementation is carried out using the C# language in Microsoft Visual Studio. Database work was engineered using Microsoft SQL Server and User defined *stored procedures*. To create the robot tasks for the case study, requires the design of an event driven and state-based behavior process, using a state machine - see Figure 4.7.

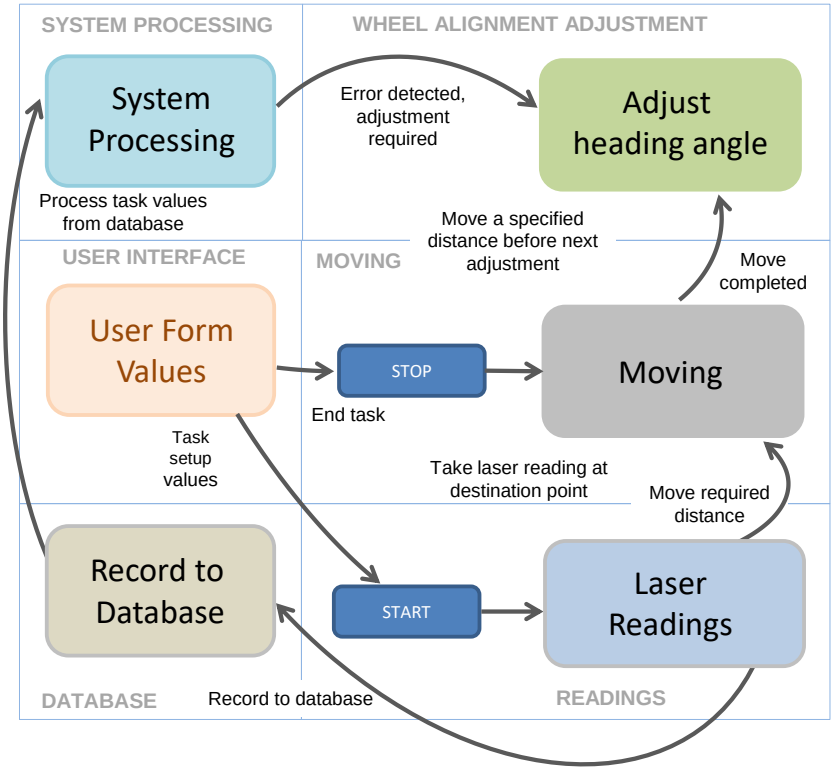


Figure 4.7: State Machine Design for the case study Wheel Alignment Error.

In the MRDS framework, as each *state* is executed, transitions are added. These transitions are triggered by notifications received from *Partner* services [95]. The System Processing 'state' (see Figure 4.7), is executed after the robot has completed a series of tasks. If necessary, the System Processing can initiate the 'Wheel Alignment Adjustment' state.

4.3.3 Knowledge Base - applied to AIFH

In IBM's Autonomic Blueprint [47], Knowledge source is described as containing different data types such as symptoms, policies, change requests and change plans.

This knowledge can be stored and shared among autonomic managers. For autonomic fault handling, *knowledge base* is important not only to store historical data but also, data such as tolerance values, real-time component data, adjustment policies and symptoms. In research conducted in [96], they use the *Knowledge Base* to store *Recovery Patterns*. When a component failure occurs, the autonomic will then select the appropriate recovery pattern(s) to compensate for the fault. Knowledge-based approaches can use SDG (Signed Direct Graph), to represent the topology of the robotics system. Faulty expressions can be associated with both hardware and software faults. These expressions can be represented as nodes such as sensors, actuators and feed-back. Each node stores a value and a limit/above within the node is considered as been faulty [97].

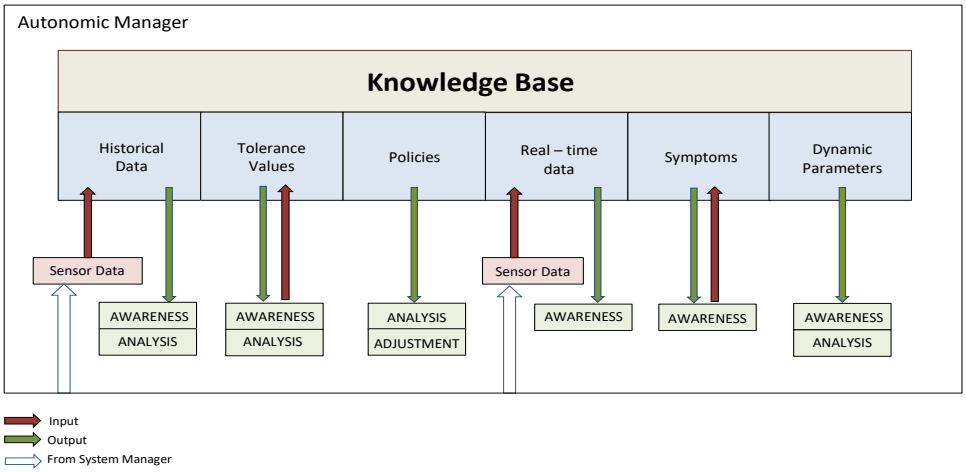


Figure 4.8: Knowledge Base - how *knowledge* is partitioned to reflect autonomic fault handling in a mobile robot.

Figure 4.8 shows how the *knowledge base* can be implemented in autonomic fault handling for a mobile robot. Sensors data (provided by the System Manager), is *filtered* by the Autonomic Manager, so only relevant information is available to the *knowledge base*.

- Real-time data - is available to the *Awareness Layer*, so that any anomalies in the data can be quickly identified.

- Tolerance Values - these values are used by the *Awareness Layer* to check

that sensor data is within threshold limits. Tolerance Values can also be updated by the *Analysis Layer* if behavior changes require threshold limits to be adjusted.

- Dynamic Parameters - used to compare a component value with a 'dynamic' value stored in the Knowledge Base. If they are equal, then action can be taken within the *Awareness Layer* or *Analysis Layer*.

- Historical Data - over time, sensor data is recorded into the Knowledge Base for evaluation. This data can be used to evaluate the performance of each sensor as the robot completes its tasks.

- Symptoms - if sensor readings are trending towards tolerance limits, then 'symptoms' data can be stored here and made available to the User Interface or Mission Control, to alert of possible impending faults. 'Symptoms' data is the responsibility of the *Awareness Layer*.

- Policies - these are specialized algorithms used to analyze data that has been flagged in the *Awareness Layer* to indicate a possible fault. The policies contained in the *Analysis Layer* can evaluate the extent of the fault. This 'evaluated' data is then passed to the *Adjustment Layer*. The policies contained in the *Adjustment Layer* are used to compensate to the fault.

4.4 Implementation

The Conceptual Design in Section 4.3 provided the architectural Autonomic Model for handling the *alignment* fault. The Implementation Section will show how the *fault* data is analyzed and how an *adjustment* algorithm can be formulated to compensate for the fault.

4.4.1 Robot Task Data Evaluation

Using User input values, the robot performs a series of tasks. The readings from each task are recorded in the SQL database. After a series of tasks are completed,

the *evaluation* algorithm is executed to establish if the data received is within tolerance limits. This *evaluation* algorithm contains a Standard Deviation equation (non-grouped data), which produces a *performance* value that can be compared to the expected *tolerance* value. Table 4.1 shows testing carried out using the P3-DX robot and LMS laser finder. Test Scenario A represent the robot operating with both drive wheels at optimal performance. Test Scenario B represents the robot operating with one damaged wheel. A 'damaged' wheel is simulated by removing some of the inner packing structure of the wheel. The means the wheel cannot support the robot fully and causes the robot to slew to one side.

Table 4.1: Pioneer P3-DX wheel alignment testing - the numbers represent the amount in millimetres (mm) that the robot was from its required destination point, after each task.

Pioneer P3DX Wheel Alignment Testing											
Test Scenario A: for a Robot with two wheels in optimal condition (tasks 1-50)											**SD**
1-10	2	9	-9	-4	-5	-7	-22	-7	-6	-5	
11-20	-8	-13	-13	11	-14	4	-13	2	-6	-6	
21-30	3	-11	-12	-10	4	-10	9	-11	-4	-2	8.57
31-40	-5	-10	-8	2	-7	-12	3	-5	-14	10	
41-50	6	-2	-7	4	-13	5	8	3	-4	7	
Test Scenario B: for a Robot with one slightly damaged wheel (tasks 51-60)											**SD**
51-60	-35	-49	-56	-61	-54	-69	-55	-64	-31	-45	12.16

Significant changes to the standard deviation (SD) value shown in Table 4.1 indicated that there was an error with the 'wheel alignment' on the Pioneer P3-DX robot. The Autonomic Manager (Monitoring) process identifies that there are significant changes in the SD values by comparing the SD Tolerance value set in the *Tolerance* database table with the SD value produced by the robot tasks. Using the Autonomic Manager, the robot is now *Aware* that there is a fault in the 'wheel alignment' functionality. Figure 4.9, shows the path of the robot over a predefined distance using (a) where both wheels are at optimal performance and (b) where one wheel which has been damaged.

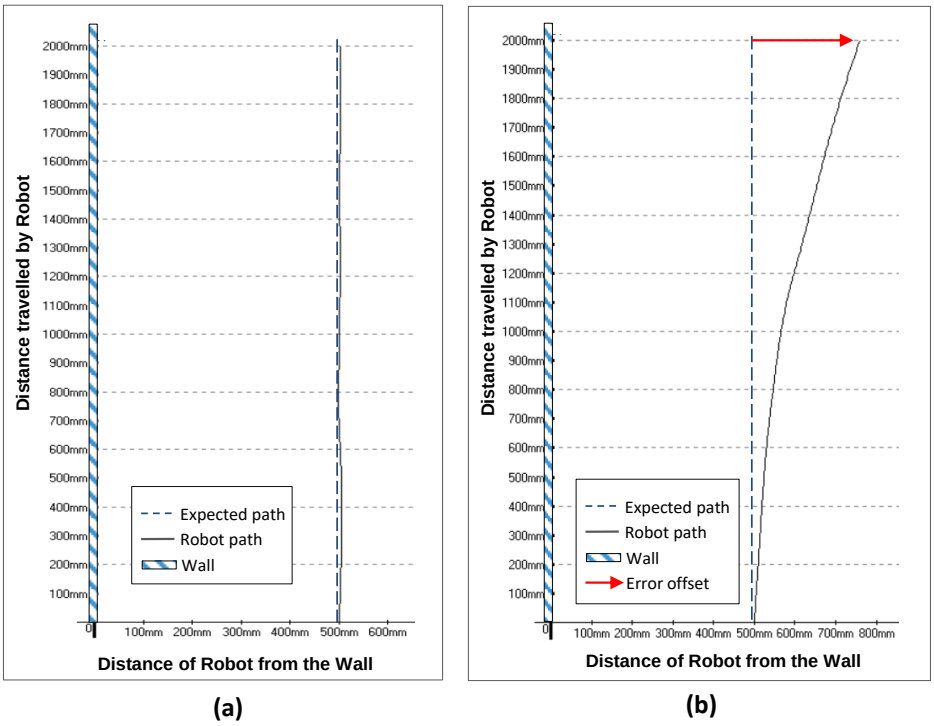


Figure 4.9: Graph (a) shows the path of the robot with both wheels at optimal performance. Graph (b) shows the path of the robot with one wheel in a damaged state.

The 'Error Offset' value can be used as part of the parameters required to calculate the *angle of error* value. The consequences of this fault are that the robot will not arrive at its expected destination point and thus will inhibit any tasks assigned to the robot. Now that the robot is slewing away from its expected destination point, to correct the alignment error, the robot will need to adjust its heading 'angle' at calculated intervals during its journey. To achieve this, the robot would travel a certain distance, stop, then turn itself back toward the expected path, then move forward again.

4.4.2 Wheel Alignment Error Evaluation

Now that an alignment error has been established in Section 4.4.1, analysis is required to define the extent of the error. In-order to *slew* the robot back towards its destination path, then the *angle error* value is required. The *angle error* value is calculated using trigonometry - see Figure 4.10

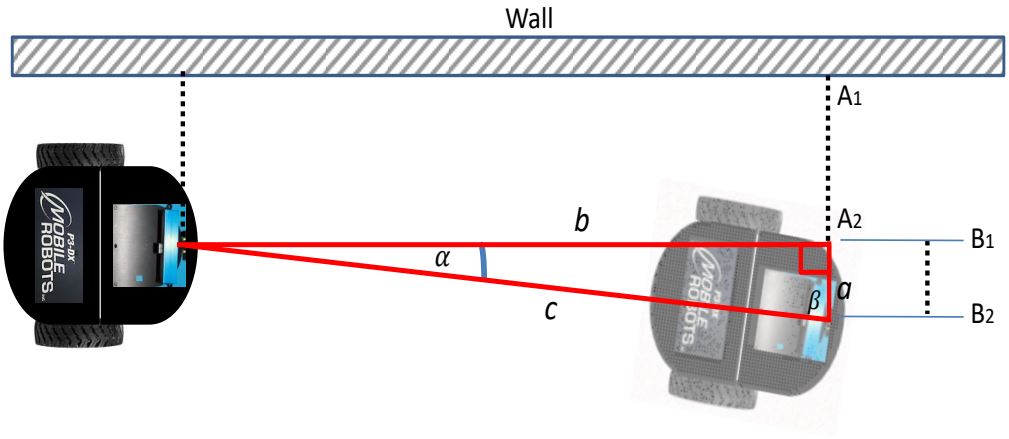


Figure 4.10: The Pioneer P3DX robot with a damaged wheel: this caused to the robot to slew to the right. A1 to A2 represents the expected distance the robot should be from the wall. B1 to B2 represents the average distance the robot was offset from the expected destination point.

Using the values take from Test Scenario B (see Table 4.1), an average distance from which the robot was from the expected destination is calculated. Using a *Right-Angled Triangle* equation (see (4.1)), the angle between the Hypotenuse side and the Opposite side is calculated (see Figure 4.10). The *angle of error* value (AE), is then used to establish the *angle of turn* needed for the robot to make its heading adjustment.

$$AE(\alpha) = \sin^{-1}\left(\frac{a}{c}\right) \tag{4.1}$$

4.4.3 Wheel Alignment Error Compensation

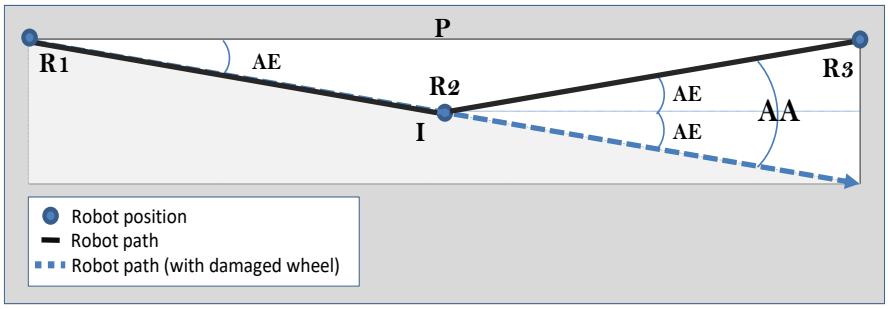


Figure 4.11: Represents how the *angle of turn* is calculated for robot alignment error compensation.

The *angle of turn* calculation is established using the values represented in Fig. 4.11. The R1, R2 and R3 represent the robot's position during its journey. When the robot reaches the position I (interval), the robot is commanded to stop. Angle AE represents the angle of the wheel alignment error calculated in Fig. 4.10. The AE angle value is then doubled. The reasoning behind this is that twice the AE values are required to bring the robot back to the expected path. The (2*AE) value is then divided by the number of intervals at which the robot is required to stop. The angle AA represents the *angle of turn* needed to allow the robot to re-establish the expected journey path marked as P. The robot *heading* angle is then adjusted, the robot is turned on its axis according to the *angle of turn* AA, see equation (4.2). The robot continues its journey by moving forward on its new heading for another interval.

$$AA = \frac{2AE}{I} \tag{4.2}$$

The more *intervals* (when the robot stops and adjusts its direction of travel), the more accurate the robot journey will be in terms of keeping to the original path. In equation (4.3), the *interval distance* is represented by ID and *total distance* is represented by TD. The interval distance is calculated as follows:

$$ID = \frac{TD}{I} \tag{4.3}$$

There is now enough data collected in Section 4.4.2 and Section 4.4.3 to compose an *compensation algorithm* for implementation. Algorithm 1, shows the robot task parameter data required (distance and number of intervals). If the robot 'error offset' value is greater than the *tolerance range* then the value of the 'angle error' is calculated. As the robot is running its tasks, the robot will stop at the predefined *interval* value and then rotate towards its original destination path. This will repeat until the robot has reached its destination point.

ALGORITHM 1: Robot Wheel Alignment Fault Compensation

Input: offsetValue = how far the robot is from expected destination point.
toleranceRange = if this value is exceeded, then an error has occurred
ni = number of required intervals
dis = distance for robot to travel
id = interval distance

Output: The angle of adjustment required ($AngleOfAdjustment$) to turn the robot when an interval.

cd = current distance traveled by robot
$id = dis/ni$
if $(offsetValue > toleranceRange)$ **then**
    $AlignmentErrorAngle$ (ae) = $sin\theta(RightAngle - equation)$;
    $AngleofAdjustment$ (aa) = $2 * ae/ni$;
end
while $cd < dis$ **do**
    Adjust the Robot direction at interval setting (id);
    if $(dis$ mod $id = 0)$ **then**
        StopRobot();
        RotateRobot(aa);
        MoveRobot();
    end
    cd = updateCurrentDistanceTravelled();
end

4.4.4 Wheel Alignment Data Trending

The *Awareness* Layer (discussed in Section 4.3.1), is not only responsible for detecting wheel alignment errors but also to highlight alignment data that may suggest that there could be possible wheel fault in the future. The Awareness Layer initiates the *Proactive* control loop to handle this process (see Fig. 4.6). If action can be taken before a fault actually occurs in real-time, then this could have a positive effect on mission objectives and resources.

Using a slightly damaged wheel on the Pioneer P3-DX, it is possible to create wheel alignment data that is significantly different to the *normal* wheel alignment data (no wheel damage) but still within tolerance levels.

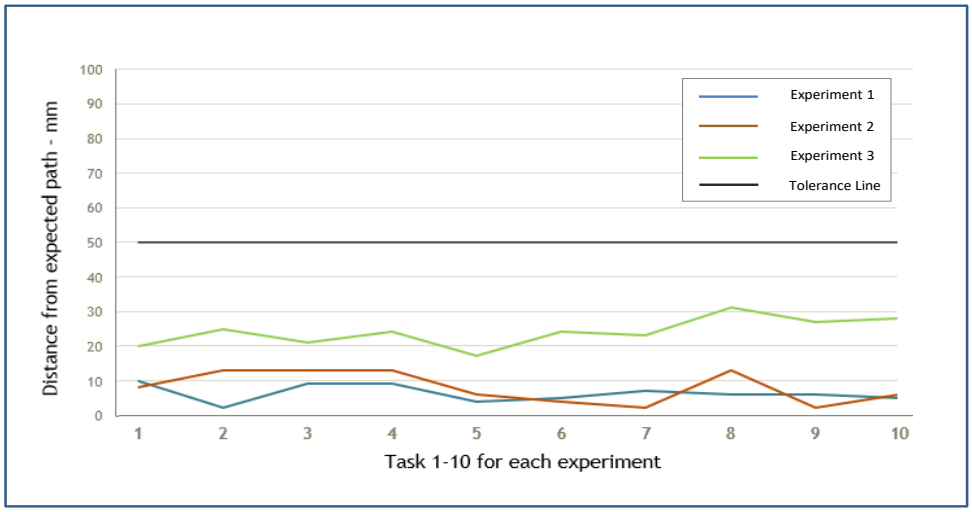


Figure 4.12: This chart shows the how wheel alignment data for a slightly damaged wheel can be used to signify a possible impending fault.

Figure 4.12 shows *wheel alignment* experiments carried out on the P3-DX robot. Experiments 1 and 2 are conducted using two fully working wheels with no reported damage. Experiment 3 is conducted with a slightly damaged wheel fitted to the robot. There are significant changes in the alignment data for experiment 3 compared to experiment 1 and 2 - but not severe enough to signify a *tolerance* fault warning.

As part of the Autonomic Manager (Awareness Layer), evaluation of this wheel alignment data provides Users (mission operations), the ability to decide if a course of action is required in terms of mission objectives or indeed to replace hardware components.

4.5 Demonstration (testing)

In Section 4.4 the implementation of the Autonomic Wheel Alignment process showed how analysis was able to calculate the extent of the alignment fault and how further calculation performed (see Algorithm 1), provided a means for compensating for the fault. This Section shows how the wheel alignment *compensation* algorithm affects the performance of the mobile robot tasks.

4.5.1 Using *intervals* in the fault compensation policy

When the mobile robot has engaged the fault *wheel alignment compensation* policy, then strategy needs to be formulated for how many *intervals* are required for the robot to complete its tasks. An *interval* is defined as the point where the robot is stopped and turned on its axis back towards its expected route path. In Table 4.2, the number of intervals will dictate the value of the Adjustment angle and the maximum offset value.

Table 4.2: Comparing offset values using a given number of intervals

Pioneer P3-DX wheel alignment testing.			
Distance to travel	Number of intervals	Adjustment Angle	Maximum offset error value
2000mm	1	12°	44mm
2000mm	2	6°	26mm

In figure 4.13, results using the *wheel alignment compensation* algorithm, show that the more *intervals* employed, then the greater the accuracy of the robot to track its expected path. This could be vital if the robot is required to drive down a narrow channel, where the *error offset* value is within the limits of the terrain. However, although a greater number of *intervals* gives greater accuracy, the more times the robot stops and rotates, then this will have an impact on resources like power consumption. Furthermore, the more *intervals* employed will also increase the time it will take for the robot to complete its tasks, as each *interval* requires the mobile robot to stop.

In Section 4.4, Test Scenario B (see Table 4.1) showed how the wheel alignment fault caused the robot to veer off course and therefore triggered the SD (standard deviation) value to rise above the tolerance value. Table 4.3 shows how the *wheel alignment compensation* algorithm has resulted in the robot now operating within tolerance values.

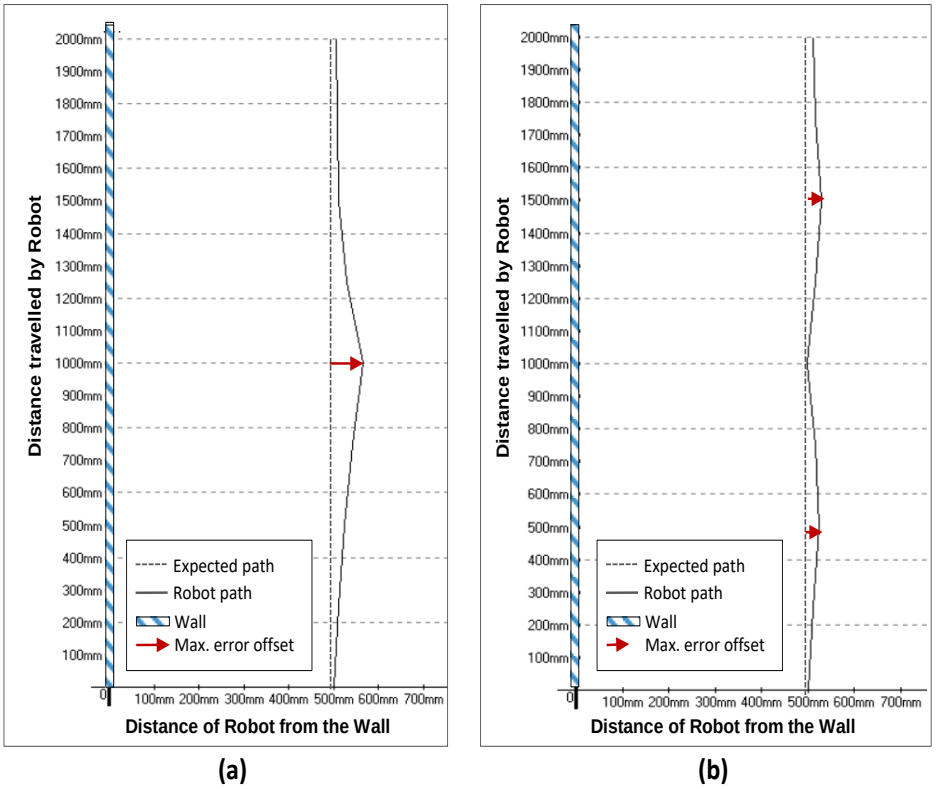


Figure 4.13: Using the wheel alignment *compensation* algorithm, the robot journey accuracy is increased when the number of intervals is also increased. (a) Robot journey uses one interval. (b) Robot journey uses two intervals.

Table 4.3: Pioneer P3DX wheel alignment testing - the numbers represent the amount in millimetres (mm) that the robot was from its required destination point, after each task.

Pioneer P3DX Wheel Alignment Testing										
Test Scenario C: a Robot with damaged wheel using compensation algorithm.										**SD**
-7	8	6	-5	4	-15	-7	3	6	-1	7.91

4.6 Evaluation

This case study posed a problem of how to detect a wheel fault on a mobile robot and then how to compensate for that wheel fault. In earlier experiments conducted around 'wheel alignment' [15], the X80 robot was used along with 'indoor' GPS to track the movements of the robot. GPS data was used to evaluate the wheel *alignment* data and algorithms where then implemented (Arc and Wave methods),

to compensate for the alignment error. The Arc Method [15], used the wheel alignment error data to point the robot in a direction that would take into account the reported *slewing error*. The Arc Method was only effective if there where no objects in the path of the robot. The Wave Method [15], used a *tacking* routine, where the robot's direction is periodically adjusted back towards the intended route path. This method allowed the robot to stay close to the expected journey path and therefore could avoid any potential obstacles. Although these methods had some success, the accuracy of the GPS was sporadic and unreliable. The wheel alignment *compensation* algorithm results showed improvement, but they where still far from the 'perfect alignment' value that was required, see Figure 4.14.

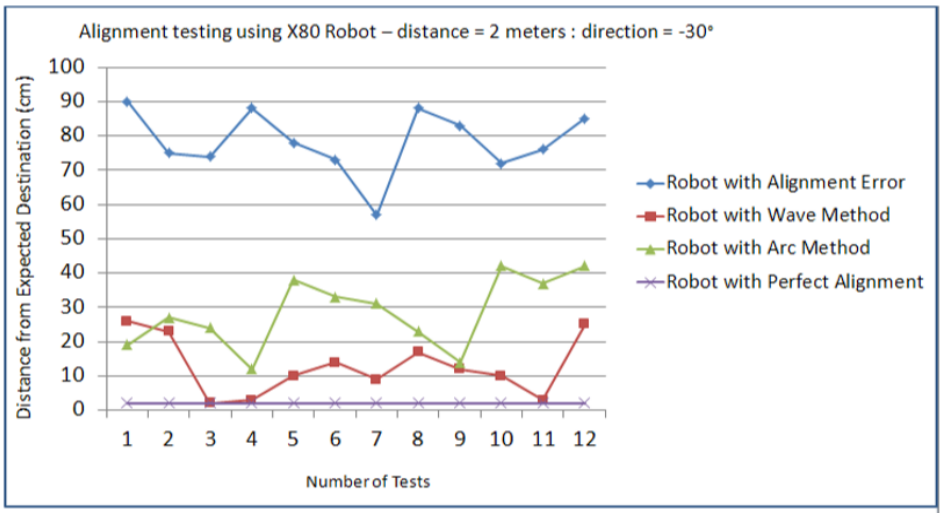


Figure 4.14: This chart shows the how the Arc Method and the Wave Method compensation algorithms improved the wheel alignment error on the X-80 robot [15].

The Pioneer P3-DX robot (equipped with the LMS 200 laser), was then introduced to the case study. The accuracy of the laser provided a reliable source for wheel alignment data. The wheel alignment experiments where first tested against the robot with no wheel damage. This was a useful experiment as it verified the accuracy of the laser readings and tested that the 'proportional drive' system on the robot was working as expected. Experiments with a damaged wheel fitted to the robot initially worked well but the main issue was that the robot was slewing considerably away from its expected path before the *compensation* algorithm could adjust

its *heading* back towards the expected path. To overcome this, *intervals* where introduced to the robot task. At each *interval*, the robot is stopped and then using the *compensation* algorithm, its *heading* is adjusted back towards the expected path. This reduced the *slewing* or *error offset* value and therefore improved how the robot followed its expected route path.

Using the AIFH architecture a *Reactive* and *Proactive* control loop was integrated into the system using an Autonomic Manager. The Autonomic Manager worked alongside the System Manager, so that the robot tasks are constantly monitored. The Autonomic Manager consisted of three layers - Awareness, Analysis and Adjustment. *Self-awareness* has the ability to detect and report on component faults within a robot. *Self-awareness* is more than just monitoring the system data. *Self-awareness* can interpret the data and inform Users of possible impending faults rather than just reacting to a 'alarm' type fault. *Self-analysis* is key to determine the extent of a fault. This information is crucial to determine if the fault can be *compensated* for. *Self-adjustment* allows the robot to continue to function even when operating with a faulty component.

4.7 Summary

The Autonomic Self-Adaptive Robot Wheel Alignment case study set out to demonstrate how an Autonomic Management System can be implemented to handle component errors within a mobile robot such as the Pioneer P3-DX. The System Development Life Cycle (SDLC) was implemented as a research model to describe a sequence of activities including requirements, design, implementation and testing. The Research question posed for this case study stated - if the AIFH architecture (derived from the MAPE-K autonomic model and IMD model) can be applied to detecting wheel alignment faults on a mobile robot and if possible, compensate for those faults? Can the mobile robot adapt to its changed environment and continue to function even with a wheel alignment fault?

The AIFH 'Knowledge Base' was used to store 'task' data from the mobile robot and to provide 'tolerance' checking against the 'wheel alignment' results.

By comparing historical data and current data, the Autonomic Manager *Awareness Layer* flagged up that there was a fault with the wheel alignment on the robot. The *Awareness Layer* is also capable of determining if the wheel alignment results, which are within tolerance limits, are significant enough to alert Users or Mission Control that there could be possible wheel fault in future robot tasks. The *Awareness Layer* then passes all wheel alignment fault data to the *Analysis Layer*. The Autonomic Manager *Analysis Layer* was then able to establish the extent of the fault and provide fault data to the Autonomic Manager *Adjustment Layer*. A *compensation* algorithm was implemented to adjust the robot 'task' parameters and therefore compensate for the fault. Although the *compensation* algorithm was implemented successfully, there where consequences regarding the operation of the mobile robot in terms of power consumption and journey time.

The research carried out in this case study provides a valuable experience in providing data in the implementation of a Generic Autonomic Architecture for fault handling in a mobile robot (see Chapter 7).

Chapter 5

Autonomic Sonar Sensor Fault Management for Mobile Robots - Case Study

5.1 Introduction

This case study is concerned with how the Autonomic Model (discussed in Chapters 2 and 3), can be applied to dealing with a sonar sensor faults on a mobile robot.

Ultrasonic range sensors are common on research robots like the Pioneer P3-DX. Sonar sensors are used to detect objects which are in the same plane as the sensor itself. This is referred to as the *2D assumption*. The sonar sensor is mounted in such a way that their acoustic axis is parallel to the floor [98]. Although traditional ultrasound sensors do not operate in a vacuum environment, they can be adapted to work on planets such as Mars, if the mechanism for generating the acoustic wave is effective in a thin atmosphere [99]. Mobile robots that operate in remote locations have a high demand on hardware reliability. Planetary rovers use both camera and sensors to navigate environment around them. Sensor failure would mean a severe impact on mission objectives. Planetary rovers such as SR2 described in [100], use range finders to help the robot detect nearby objects. Traditional fault tolerance techniques have been employed by researchers in [101], using *adapter sensor analysis*. Further studies have shown that by comparing the *known* state and the

actual sensor feedback of a collections of sensor nodes, can lead to the detection of single sensor drop-outs. If sensor failure is identified, then compensation could be possible by using *known* values instead of measured ones [102]. Detection of abnormal behaviour in sensors can also be achieved by comparing sensor data with neighbouring sensor data [79]. In this research, the authors take input readings of the sensors and subject them to a correlation test that determines which sensors are correlated to each other. In this case study, the data from suspected sonar sensors is checked by using adjacent sonar sensors. If the results between the sonar sensors do no match, then the sensor is flagged as requiring detailed analysis. Although previous research has been conducted on the performance of sonar sensors, no published research has dealt with the consequences of losing sonar sensing ability. If a mobile robot loses part of its object detection ability, then research is needed to investigate how it is possible to compensate for the loss of some of the sonar sensors.

If a fault occurs in a sensor, then there is no realistic way of retrieving the remote robot for to repairs. For mobile robots to navigate within their environment, they rely on *object* detection sensors. When an object is detected, then the mobile robot can adjust its drive system to avoid the object. Types of sensors used to detect objects are sonar, laser and camera sensors. For the purpose of this case study, sonar sensors are utilized for experimental purposes. If a sonar sensor on a mobile robot becomes faulty, then the robot's ability to detect objects is greatly reduced. This case study explores how detection of a faulty sonar sensor(s) is achieved and how *self-adaption* can be implemented to compensate for the fault.

In Chapter 4, the basic AIFH architectural design was implemented to handle wheel alignments faults in mobile robots. However, the Knowledge Base was not well defined as there was no named attributes such as *policies*, *real-time data* and *tolerance values*. In this case study, the Knowledge Base is clearly defined as it presents the role each *attribute* plays and how they contribute to handling sonar sensor faults in mobile robots.

This case study is organized as follows: Section 5.2 describes how Research Methods are achieved by utilizing the SDLC (Software Development Life Cycle)

modal. Section 5.3 presents Conceptual Requirements which describe the Research question, goals and resources needed to complete the case study. Section 5.4 presents the Conceptual Design and describes how the Autonomic Management System is used to *self-adapt* when dealing with mobile robot sonar sensor faults. Section 5.5 presents Implementation of the wheel alignment fault scenarios and how Awareness, Analysis and Adjustment can used to evaluate and compensate for a sonar sensor fault. Section 5.6 demonstrates through testing, how the *compensation* algorithm for the sonar sensor fault performs. Results from the testing are then analysed. Section 5.7 is used to evaluate the results in the case study and finally, Section 5.8 concludes the case study with a summary statement.

5.2 Research Method

Research Methods are based in the SDLC (Software Development Life Cycle) model - see Figure 5.1. Each stage in the SDLC is used to develop the *research question*, design the autonomic architecture used in the Sonar Sensor Fault Manager, implement specialized algorithms to identify and compensate for the fault and testing to verify the case study goals are achieved.

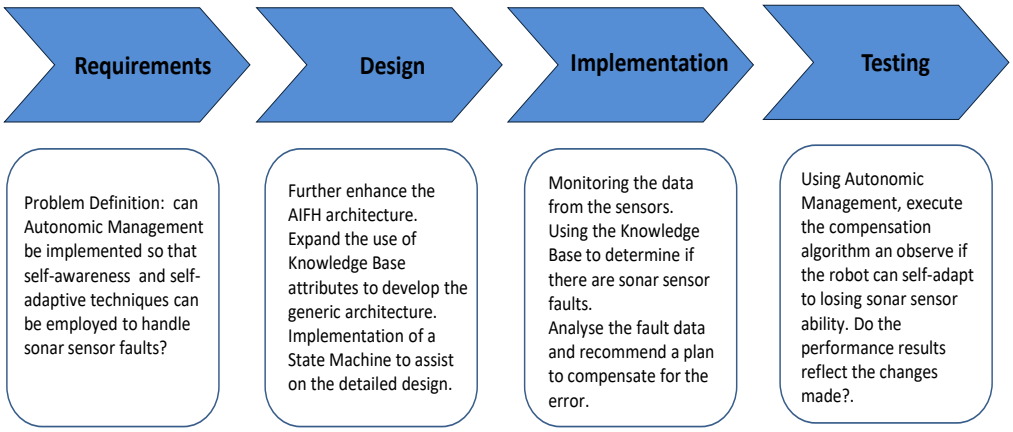


Figure 5.1: SDLC Model used in the research methodology for Sonar Sensor Fault Management.

- **Requirements** - is a detailed investigation of the system and is carried out in accordance to the objectives proposed. The requirements are concerned with

defining research question and what resources are required to complete the case study. To develop the research question, physical components such as the Pioneer P3-DX robot fitted with a Sonar Sensor forward array is required. A 'bumper' component is also required to prevent the robot being damaged when coming into contact with an object.

- **Design** - based on the information collected at the requirements stage. For this case study, the design is concerned about applying the Autonomic Architecture to handle how the robotic system can *self-adapt* to the discovery of a component fault. The design phase will incorporate the AIFH architecture (Awareness, Analysis and Adjustment) and further develop it to handle the Sonar Sensor fault scenario. The design phase will also incorporate a State Machine. The State Machine implemented using MRDS and CCR (Concurrency Co-ordination Runtime) [58], which can handle recording of sensor data, analyses of sensor data and initiate sensor fault adjustment if required.

- **Implementation** - the system design needs to be implemented to make it a workable system. To create the sonar sensor fault experiment, each sonar on the sensor array is tested. Readings from each sensor are verified against measurements taken with a physical tape measure. Adjacent sonar sensors are tested to check for correlation. Implementation includes designing custom algorithms to identify multiple sonar sensor fault scenarios. Finally, a specialized algorithm will be required to compensate for sonar sensor faults. In this case study, the AIFH architecture is employed: *Awareness* - to identify any faults with the sonar sensors, *Analysis* - to evaluate the extent of the fault discovered and finally *Adjustment* - to implement a compensation strategy to deal with sonar sensor faults.

- **Testing** - involves system integration and system testing of the programs and procedures coded at the implementation phase. Testing is a method of testing the system against the requirements and design. In this case study, testing will involve verifying the sonar sensors are working as expected. Testing

the Analysis programming so that faults are identified and finally testing the compensation algorithm can still provide the robot the ability to detect an object even with some sonar sensor faults. Test Strategy:

1. Test that the sonar sensors are reporting the correct distance readings.
2. Do neighbouring sonar sensors report similar distance readings?
3. The Pioneer P3-DX robot will report a default sensor reading of value 5000, if a sonar sensor becomes unresponsive. If a sonar sensor reading equals a value '5000', the robot will halt its current task.
4. Does the *Analysis Layer* knowledge-based policies indicate what sonar sensors need to be flagged as being disabled?
5. Does the *Adjustment Layer* knowledge-based policy (compensation algorithm), allow objects to be detected even when the robot has reported 'disabled' sonar sensors?

5.3 Conceptual Requirements

5.3.1 Problem Definition

5.3.1.1 The Experiment

Using the Pioneer P3-DX robot fitted with a Sonar sensor array: in laboratory conditions, the sonar sensors on the array are tested to confirm that the distance readings they report are accurate when measured using traditional measuring tape. Adjacent sonar sensors are also tested to check that their readings are accurate. The main experiment involves sonar sensors that are *disabled* and how the mobile robot can compensate for the loss of its *object* detection ability.

5.3.1.2 Goals

Can the AIFH architectural model be further developed to detect/compensate for faulty sonar sensors?

- **Awareness** - can monitoring past and present experimental data, allow the Autonomic Manager the ability to decide if there is a sonar sensor fault?

- **Analysis** - can analysis provide the means to establish the extent of the fault?
- **Adjustment** - can the fault data provided by analysis, be used by a *knowledge base* policy, to provide a means to compensate for the fault in the sonar sensor array?

5.3.2 Resources required

For this case study, the Pioneer P3-DX research robot is used. The P3-DX is equipped with an array of Polaroid sonar sensors. The array comprises of 8 electrostatic transducers and a sonar *ranging* module - see Figure 5.2 (b). The individual transducers are controlled by the *ranging* module. The 'echo' signals captured by the transducers, allows the module to calculate ranges from 6" to 35ft [93]. For the purposes of this case study, only the 6 forward facing sonar sensors will be used in the experiment - see Figure 5.2 (a). Each sonar sensor is placed on the array as part of an octadecagon shape - see Figure 5.3.

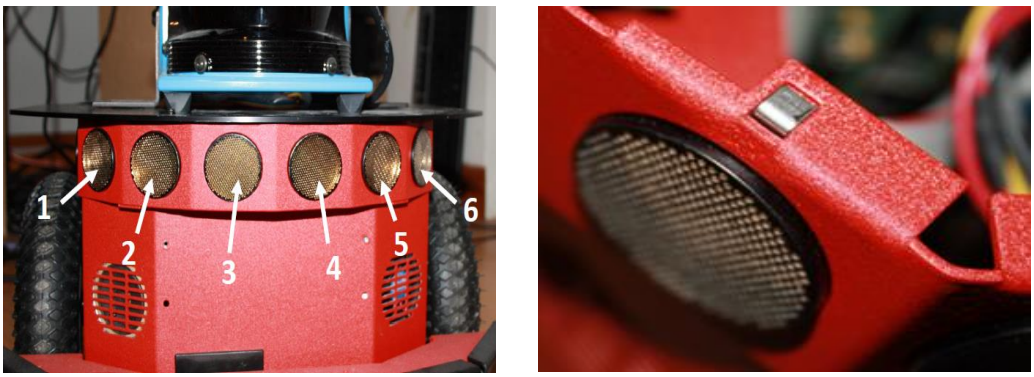


Figure 5.2: (a) P3-DX with its 6 forward facing sonar sensors. (b) Each sensor comprises a Polaroid transducer.

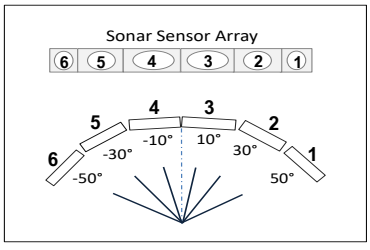


Figure 5.3: The sonar sensors are arranged 1-6 on the array, with a 20 ° angle between each sensor.

To access the data reported from each sonar sensor, a custom-built User Sonar Interface was required. The User Sonar Interface was developed using Microsoft's MRDS and .Net C# programming language in Microsoft Visual Studio - see Figure 5.4. The User Sonar Interface can display *distance* readings for each sonar on the sensor array mounted on the P3-DX robot - Sonar 0 to Sonar 7. Any *disabled* sonar sensors are marked in red on the interface display. The P3-DX robot can also be controlled using the User Sonar Interface for forward/backwards movement and rotating.

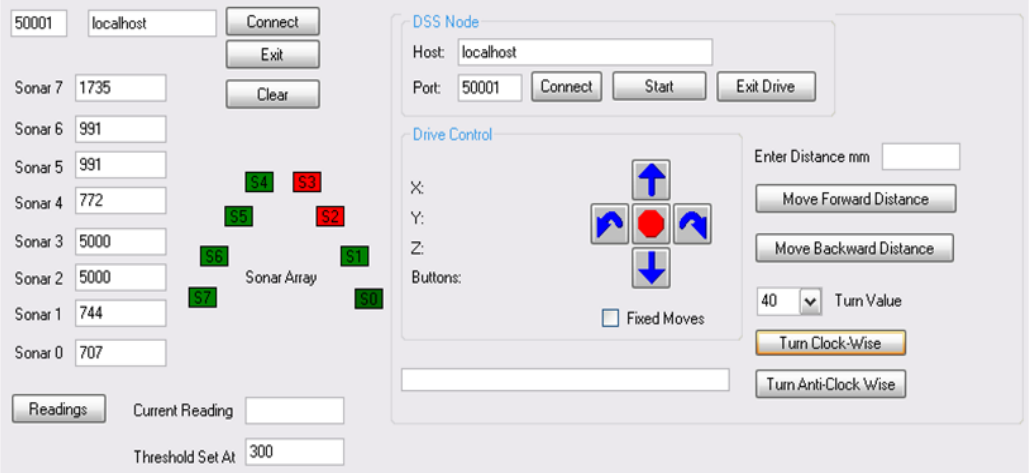


Figure 5.4: User Sonar Interface (developed by the author), for displaying sonar data from the Pioneer P3-DX robot.

5.4 Conceptual Design

This Section details the design and architecture for this case study.

5.4.1 Developing the AIFH Architecture for sonar sensor fault handling

In Chapter 4, a case study showed how the AIFH Architecture was developed for handling *wheel alignment faults in a mobile robot*. In this case study, the AIFH architecture will be further developed for detected Sonar Sensor Faults in a mobile robot, with particular emphasis on the use of the *Knowledge Base*. This Autonomic Manager contains three layers, Awareness, Analysis and Adjustment. The System Manager is responsible for initiating the 'control' loops. The main feedback control

loop (Reactive Loop), will feed data through each layer. The secondary control loop (Proactive Loop), will operate in the Awareness and Analysis Layer. The AIFH architectural model for the Sonar Sensor Fault handling is shown in Figure 5.5.

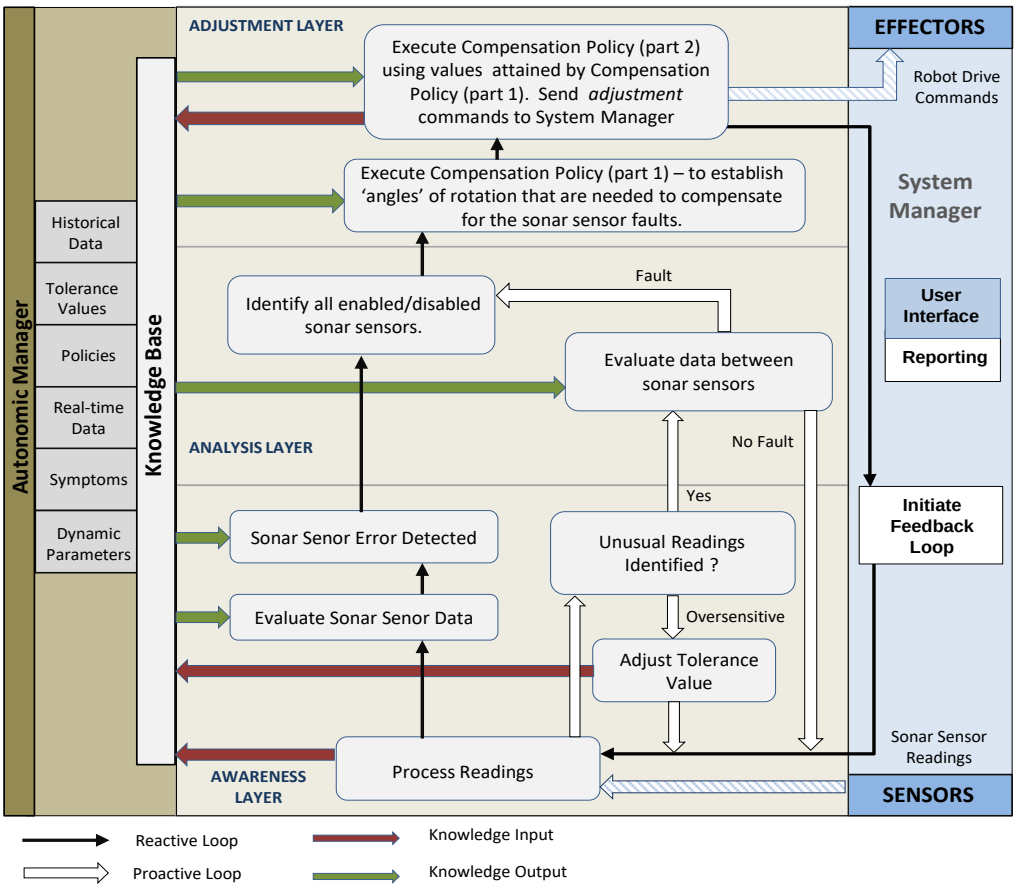


Figure 5.5: Autonomic Management System for the Sonar Sensor Fault Handling case study.

1. *Awareness Layer* - the main function of the Awareness layer is to establish if there is a failure within the sonar sensors. If a failure is detected, then the data gathered within this layer is passed to the Analysis layer. The Awareness Layer can also detect if there are unusual readings between adjacent sonar sensors. Those sonar sensors that are showing unusual readings, are flagged as requiring investigation. Task data from the sonar sensors is processed and updated to the Knowledge Base. As each task is performed by the mobile robot (Pioneer P3-DX), the sonar sensor data is recorded. These records will then collate to form the *historical data* within the Knowledge

Base. The Knowledge Base *Real-time* sonar data is compared to the stored *Dynamic Tolerance value*, which will in turn, identify if any sonar sensors are *disabled*. The Pioneer P3-DX robot reports a default reading of value 5000, if a sonar sensor is unresponsive (this value is stored in the Knowledge Base as a *dynamic tolerance value*). If a sonar sensor reports a reading of '5000', then it is immediately marked as being disabled. The Reactive control loop is responsible for reporting 'disabled' sonar sensors to the Analysis layer.

The Proactive Control loop is responsible for evaluating readings between adjacent sonar sensors. If the robot is facing an object, then each of the sonar sensors will report a slightly different reading to its adjacent sensor. This is due to the fact that the sonar array fitted to the robot is octadecagon in shape - see Figure 5.3. The Awareness Layer must establish what the current *tolerance range* value is between adjacent sonar sensors. A policy from the Knowledge Base is used to calculate the *tolerance range* value. This will be explained in more detail in Section 5.5.3. This policy is enforced to prevent the Proactive Control loop being 'oversensitive' in declaring a possible fault, if readings between two adjacent sonar sensors are not the same. The *tolerance range* is dynamic, and therefore can be updated in the knowledge Base by the Proactive control loop. However, if readings between adjacent sonar sensors is above the acceptable *tolerance range*, then the Proactive Control loop will mark these sensors as 'suspect'. Data relating to 'suspected' sonar sensors is passed to the Analysis Layer for further investigation.

2. *Analysis Layer* - the Analysis Layer uses data received from the Awareness Layer to establish the extent of the sonar sensor failure. This layer will map out what sonar sensors have been declared as 'disabled' and pass this information to the Adjustment layer. The Analysis layer will also check sonar sensors that have been identified in the Awareness layer as requiring further investigation. When a sonar sensor is reporting a different reading from its adjacent sensors, then analysis is required to established that the sensor is operating correctly. The *Check Sonar Reading* policy (from the *knowledge base*), is

then executed. This policy (algorithm), uses neighbouring sonar sensors to verify that the suspected sonar sensor is indeed reporting valid data. If it is established that the sonar sensor is reporting invalid data, then that sonar sensor is marked as being disabled. If, however, adjacent sonar sensors are reporting valid data, then the Proactive control loop will declare a 'no fault' and report back to the Awareness Layer.

3. *Adjustment Layer* - the Adjustment layer receives data from the Analysis layer that identifies what sonar sensors are currently disabled. Depending on the number of sensors that are marked *disabled*, will influence the amount of adjustments required to handle the fault. First, the Adjustment layer needs to establish how many sonar sensors are *disabled*. If all sonar sensors are *disabled* then no adjustment policy can be applied in-order to compensate for the fault. If there one or more sonar sensors are still functioning, then the Adjustment Layer will execute Compensation policy (part 1), to establish how many *rotations* the robot will have to make to compensate for the *disabled* sensors. This is explained in greater detail in Section 5.5.3. When all the 'rotation' angles have been calculated, this data is then passed to Compensation policy (part 2). Compensation policy (part 2), will then execute 'rotation' commands via the System Manager, in-order for the P3-DX robot to locate any objects in its path during while executing a task - (see Figure 5.5). All *compensation* polices are stored in the Knowledge Base. The Knowledge Base is also updated with the current state of each sonar sensor.

5.4.2 State Machine

Software development for this case study was carried out using MRDS framework. MRDS is a service-oriented programming model that allows the creation of asynchronous and state-driven applications [94]. Code implementation is carried out using the C# language in Microsoft Visual Studio. Database work was engineered using Microsoft SQL Server and User defined *stored procedures*. To create the robot tasks for the case study, required the design of a event driven and state-based

behavior process, using a state machine - see Figure 5.6.

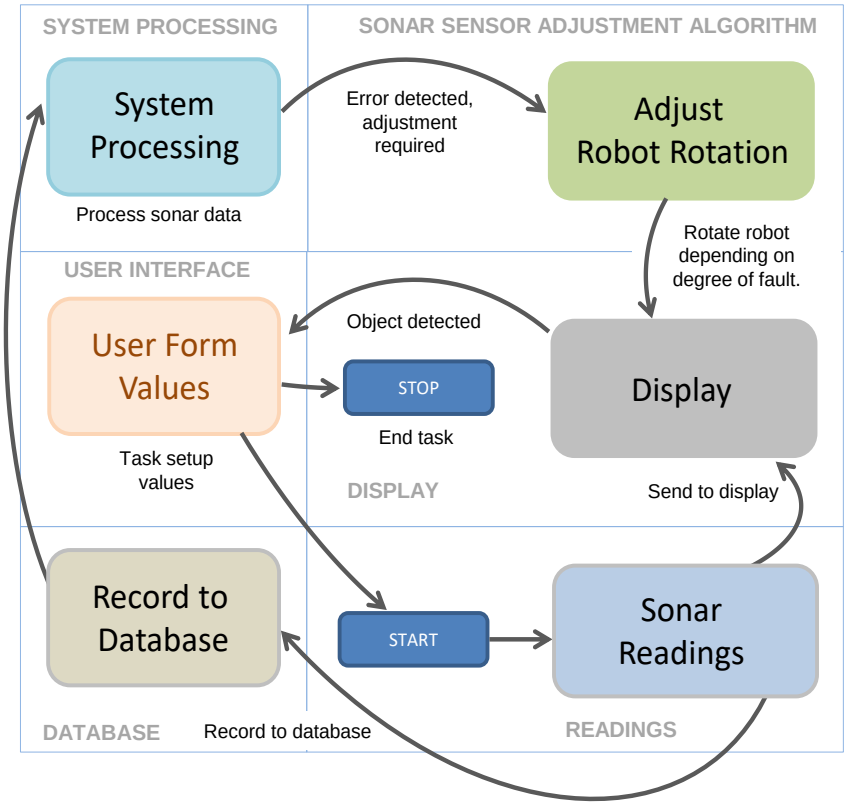


Figure 5.6: State Machine Design for the case study - Sonar Sensor Fault Management.

In the MRDS framework, as each *state* is executed, transitions are added. These transitions are triggered by notifications received from *Partner* services [95]. The System Processing 'state' (see Figure 5.6), is executed after the robot has monitored and analysed the sonar sensor data. If necessary, the System Processing can initiate the 'Adjust Robot Rotation' state.

5.5 Implementation

5.5.1 Sonar sensor fault Scenarios

Ranging sensors like sonar, are widely used in research and industrial robotics. They allow a robot to see an object without actually coming into contact with it. However, sonar sensors are limited to a relatively low range distance compared to a sensor such a a laser. Sonar sensors can also suffer from 'Ghost' echoes, where

there is a dense obstacle distribution and complex surfaces [103]. When a sonar sensor becomes faulty it can impact a robot's ability to navigate its surrounding environment. A *single* sonar sensor fault would result in a minor reduction in the robot's ability to detect objects as its neighbouring sensors can compensate for the failure. However, if the one or more sonar sensors fail, then this severely impacts the robots object detection abilities.

5.5.2 Sonar Sensor Failure States

Sonar sensors faults in this case study are classed as *failure states*. Each *failure state* represents a fault level that can occur on the P3-DX mobile robot. Sonar Sensor Failure States (see Figure 5.7):

- IsNormal - all sonar sensors are working as expected - Figure 5.7(a) .

- IsMinor - one or two sonar sensors are either disabled or reporting erroneous data - Figure 5.7(b).

- IsMajor - a loss of 3 or more (but not all) sonar sensors. Provides only limited sensing ability - Figure 5.7 (c).

- IsCatatrophic - all forward facing sonar sensors are disabled. No ability to detect objects - Figure 5.7 (d).

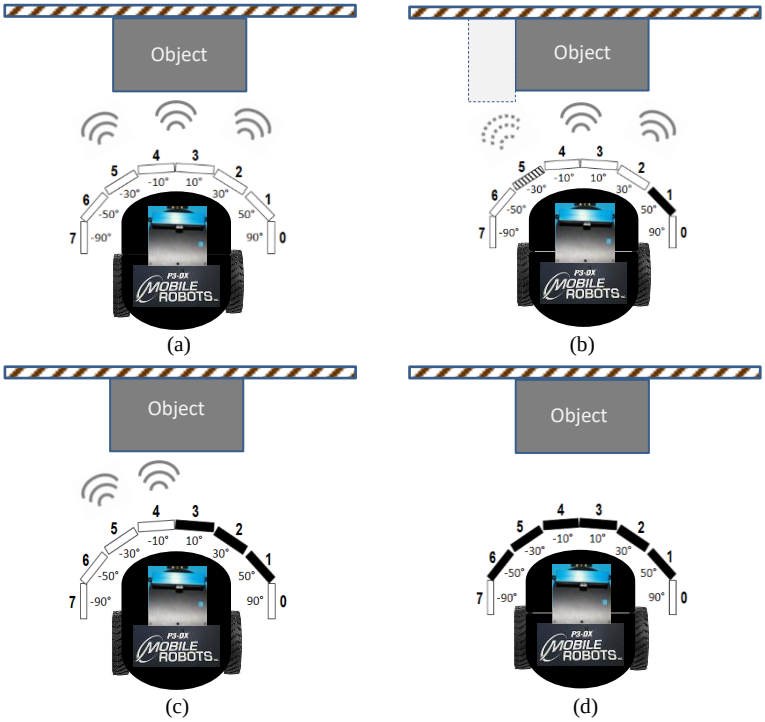


Figure 5.7: Failure states for the sonar sensors on the P3-DX mobile robot.

5.5.3 Detecting Sonar Fault - *Awareness*

Using the architectural model discussed in 5.4.1, the Awareness Layer is part of the process that detects any anomalies within the sensors. Through monitoring and *knowledge* gained from previous robot tasks; the robot can become *Aware* that there is a possible fault with the sonar sensors. Using the *failure states* described in 5.5.2, various test scenarios where performed to establish if there was a possible fault within the P3-DX sonar array.

5.5.3.1 IsNormalState

All sonar sensors where tested under normal conditions. Using the User Sonar Interface program, the *IsNormalState* (see Figure 5.7 (a)), proved that each of the sonar sensors where able to detect objects correctly. Measurements where taken between the object and each sonar sensor using a measuring tape - see Figure 5.8 (b) and (c). These values where then compared to the values being reported by the sonar sensors using the User Sonar Interface program (discussed in Section 5.3.2) -

see Figure 5.8 (a).

Figure 5.8: IsNormalState Test - checking that the sonar readings reported by the robot are accurate.

5.5.3.2 IsMinorState

When a sonar sensor becomes faulty (due to impact or electrical issues), then it reports via the System Manager, as a value of 5000 (*disabled state*). Autonomic Manager (Awareness process) will pick this value up during the *Reactive* control loop check. If a sonar sensor reports a reading of '5000', then the sonar sensor is marked as *disabled*. If a sonar sensor is reporting significantly different 'distance' data to its neighbouring sensors, then an assessment is required to confirm its validity. For example: on the robot sensor array, Sonar 4 is reporting a value of '415', Sonar 5 a value of '245' and Sonar 6 a value of '417'. Sonar 5 needs checking as its value is considerably lower than Sonar 4 and Sonar 6. However, Sonar 5 could be detecting an object and reporting a correct reading; this can be verified by using the adjacent sensors to check the reading is valid, see Figure 5.10 (object marked as *d*). When comparing the values of neighbouring sonar sensors, we need to consider the location of the sonar sensors within the sonar array. The forward- facing sensors on the sonar array (1-6), are arranged as part of an octadecagon design. If a sonar sensor is detecting an object *square-on*, then its neighbor senor will also detect this object but will be reporting the distance value as slightly higher - see Figure 5.9. This *difference* value needs to be considered when comparing neighbouring sonar sensors. Figure 5.10 shows how the *tolerance range* is calculated between neighbouring sonar sensors. Equations 5.1 and 5.2 used to calculate ***tr*** the *tolerance* value. These calculations contribute to the logic used in Algorithm 2.

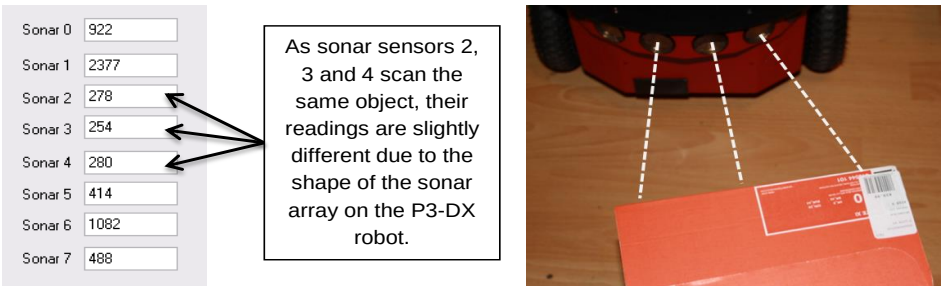


Figure 5.9: IsMinorState Test - readings for adjacent sensors are slightly different to the octadecagon design of the sonar array on the P3-DX robot.

$$b = \left(\frac{a1 + a2}{Cos(\alpha)} \right) \tag{5.1}$$

$$tr = (b - a1) - (a2) \tag{5.2}$$

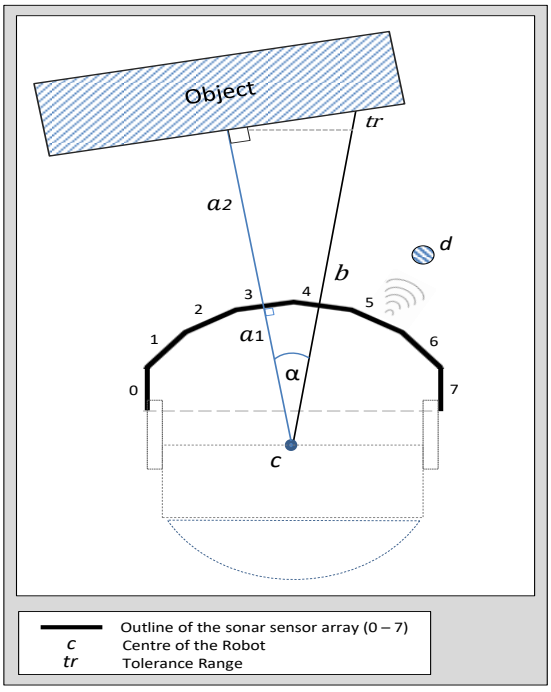


Figure 5.10: The *difference* value between two adjacent sensors is calculated to allow for the octadecagon design of the sonar array. This is described as the *tolerance range* ***tr***

.

ALGORITHM 2: Disparate Readings Between Adjacent Sonar Sensors

Input: sonarReadings sr[] = readings from 1-6 sonar sensors
toleranceRange tr = tolerance value allowed between adjacent sensors
sonarPostion sp = position of specific sonar sensor
Rotation Angle ra = 20°

Output: *differenceValue* = is greater than the tolerance range, then that particular sonar sensor is marked as disabled.

for *(each sonar(sn) in sonar array)* **do**
  if *(sn == 1)* **then**
    $reading1$ = sr[sn];
    $RotateRobot(-ra)$;
    $reading2$ = sr[sn+1];
    $differenceValue = (reading2 - reading1)$;
  end
  if *(sn == 6)* **then**
    $reading1$ = sr[sn];
    $RotateRobot(ra)$;
    $reading2$ = sr[sn-1];
    $differenceValue = (reading2 - reading1)$;
  end
  if *(sn > 1 and sn < 6)* **then**
    $reading1$ = sr[sn];
    $RotateRobot(ra-)$;
    $reading2$ = sr[sn+1];
    $RotateRobot(ra + ra)$;
    $reading3$ = sr[sn-1];
    $differenceValue = (reading1 - (reading2 + reading3/2))$;
  end
  if *(differenceValue* $> tr$*)* **then**
    $sn = disabled$;
  end
end

Algorithm 2: - a tolerance range ***tr*** value can now be applied to Algorithm 2, where all the sonar sensors are checked for any unusual values. Adjacent sonar sensors are rotated towards the same target. Sonar *distance* readings from each sensor are recorded. Logic contained in Algorithm 2, can identify what readings are significantly different from their immediate neighbours. The Awareness Layer has access to the Algorithm 2 policy via the Knowledge Base. If a sonar sensor requires checking, then this information is passed to the Analysis Layer for processing.

5.5.3.3 IsMajorState

When two or more sonar sensors become faulty (see Figure 5.7 (c)), then the robot's ability to detect objects in its path is greatly reduced. If the robot loses 50 percent of its sonar sensors, it can be completely blind on one side. Monitoring of the sensor data would indicate that there was a fault in several sonar sensors in the array i.e., each of the faulty sonar sensors would be reporting a '5000' value (*disabled state*).

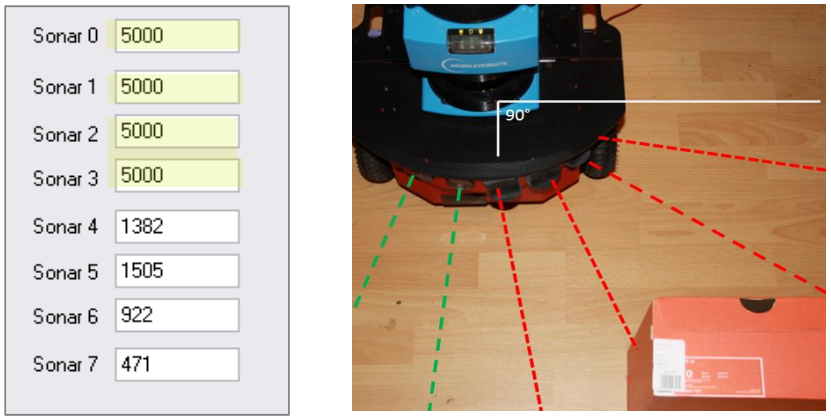


Figure 5.11: The User Sonar Interface program (a), shows Sonar sensors 0-3 with a '5000' reading = *disabled* state. The object in (b), cannot be detected.

In Figure 5.11, the mobile robot has lost sonar sensors 0 - 3. The object ahead of the robot is currently undetected. The if the robot continues this trajectory it will hit the object. However, the P3-DX is also equipped with a 'bumper' sensor. If the 'bumper' sensor is triggered, then the robot automatically comes to a stop. When a sonar sensor becomes disabled, the autonomic *Awareness* process is employed to identify what sonar sensors are faulty (see Figure 5.5). The the *disabled state* '5000' value is stored in the Knowledge Base as a 'fixed' tolerance value. As the *Awareness* layer processes the sonar sensor data, it can use this 'fixed' tolerance value in the Knowledge Base, as a trigger to detect a faulty sensor. All 'fault' data is then passed from the *Awareness* layer to the *Analysis* layer for further processing. The *Analysis* layer will then determine the extent of the fault using policies from the Knowledge Base.

5.5.3.4 IsCatastrophicState

In Figure 5.12, the *Awareness* layer reports that all sonar sensors are disabled. Using the Knowledge Base 'fixed' tolerance value for a *disabled state* '5000' sonar sensor, the *Awareness* layer is able to determine (become aware), that the robot has no *object* detection sensors available. When all sonar sensors are reported as disabled, the robot is automatically stopped by the System Manager. If all the sonar sensors in the array become disabled at the same time, the P3-DX 'bumper' sensor (see Figure 5.12(c)), will also stop the robot from further movement; this is to prevent any unnecessary damage to the body of the robot. In this state, the robot is unable to detect objects in its path. No compensation algorithm can be applied when the robot is in this state.

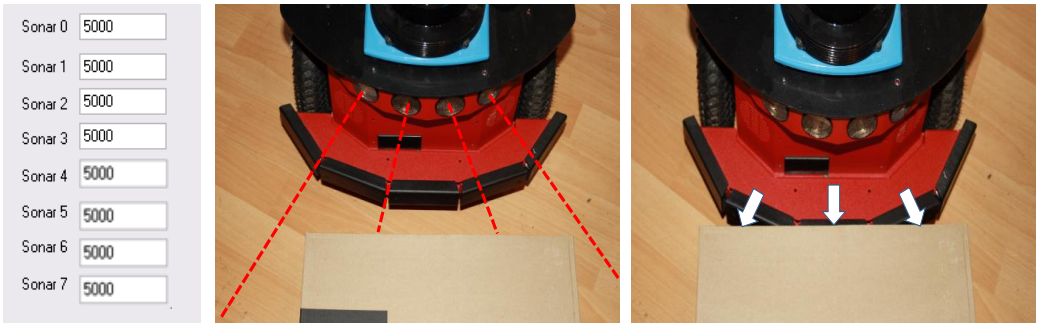


Figure 5.12: IsCatastrophicState - all the sonar sensors are reported as 'disabled' (a). The sonar sensors can no longer detect objects in the path of the robot (b). The P3-DX 'bumper' sensor will cause the robot to stop, when coming into contact with an object (c).

For testing the different sonar sensor *states*, the User Sonar Interface program, was developed for this purpose. Many hours where spent by the author to develop a software solution, that converted the raw data from the sonar sensor device drivers, into readable data and presented as a User Interface (see Appendix B2).

5.5.4 Processing Sonar Fault - *Analysis*

If during the Awareness Layer, a Sonar Sensor (sonar[x]) is identified for checking, then this data is passed to the Analysis Layer for processing. In the Analysis Layer, the (*Check Sonar Readings*) policy from the Knowledge Base, is then executed - see Algorithm 3. The *Check Sonar Readings* algorithm will issue commands to the robot to use the sonar sensors adjacent to the sonar[x], to check if the readings reported by sonar[x] where indeed correct.

Algorithm 3: - the process performed by Algorithm 3, involves using two neighbouring sonar sensors to be rotated to the original position of *sonar[x]* - see Figure 5.13. If the readings reported by both sonar sensors are different from *sonar[x]*, then *sonar[x]* will be tagged as being disabled. If *sonar[x]* is at position one or six on the sonar array, then it will have only one neighbouring sonar sensor available for checking. If the reading reported by this one sonar is different from *sonar[x]*, then *sonar[x]* will be tagged as being disabled. All sonar sensors tagged as being disabled will be handled by the Adjustment Layer.

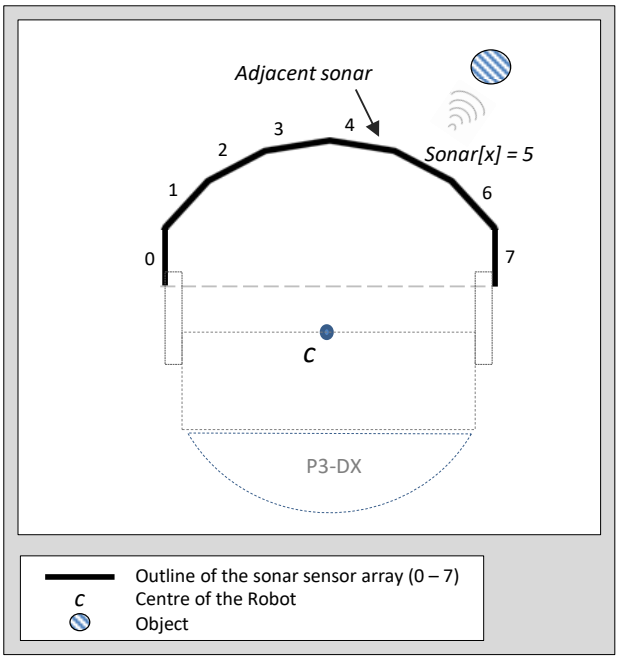


Figure 5.13: Sonar[x] has been flagged for checking by *Awareness layer*. Adjacent sensors Sonar (4) and Sonar (6) are used to verify that reading from Sonar (5) is correct.

.

ALGORITHM 3: Check Sonar Readings

Input: sonarCheck[][] contains the sonar position and readings
sr[] = current reading from the sonar sensor array
sc = 0 - track each sonar processed
col = 0 - array column track
tr = tolerance range calculated in Algorithm 2
ra = 20° Rotation Angle

Output: sonarCheck[][] is marked disabled if error is found.

for *(sc < number of sonarCheck rows)* **do**
  if *(sonarCheck[sc][col] == 1)* **then**
    $RotateRobot(-ra)$;
    checkReading = sr[sc+1];
  end
  if *(sonarCheck[sc][col] == 6)* **then**
    $RotateRobot(ra)$;
    checkReading = sr[sc-1];
  end
  if *(sonarCheck[sc][col] > 1 and sonarCheck[sc][col] < 6)* **then**
    $RotateRobot(ra-)$;
    SonarReadingA = sr[sc+1];
    $RotateRobot(ra)$;
    SonarReadingB = sr[sc-1];
    checkReading = $((SonarReadingA + SonarReadingB)/2))$;
  end
  $diffValue$ = (*checkingReading* - *sonarCheck*$[sc][col+1]$)
  if *(*$diffValue$ *>tr)* **then**
    sonarCheck$[sc][col]$ = *disabled*;
  end
  sc = sc + 1
end

5.6 Demonstration (testing)

5.6.1 Compensation for Sonar Fault - *Adjustment*

5.6.1.1 Fault scenarios

If the Analysis Layer has identified a fault in the sonar sensor array, then the Adjustment Layer will implement a compensation policy. The *compensation policy* deals with fault found in the six forward facing sensors only (see Figure 5.2). The *compensation policy* will adopt a 'stop' and 'rotate' strategy in-order to compensate for faulty sonar sensors. A fully operational sonar sensor will be rotated to position in the array of a faulty sonar sensor. The more sonar sensors that are disabled, then the more 'stop' and 'rotation' commands will be required in order for the robot to detect objects in its path.

Using the six forward sensors in the array, there are 64 possible combinations of sonar values (enabled/disabled), using binary notation. Binary notation 000000 describes all sensors are working as designed (no action required). Binary notation 111111 means all sensors are disabled (the robot is unable to detect an object - IsCatastrophicState); this leaves 62 possible sonar fault combinations that can be dealt with using the *compensation policy*.

Table 5.1 shows an example of how much the robot needs to rotate (clockwise or anti-clockwise), in order to compensate for the loss of one or more sonar sensors. There is an angle of + or - 20° between each sonar sensor on the array. A single sonar fault will only require one rotation of the robot to compensate for the missing sensor. A loss of three or more sonar sensors could require the robot to rotate at three different stages in order to compensate for the missing sensors. It must be noted, that if the robot is required to rotate a certain *degree* value to compensate for a disabled sonar sensor, after the sonar reading has been checked, the robot will be rotated back to its original position. This guarantees that the robot is always pointing to its original heading angle.

Figure 5.14, shows a representation of Scenario 4 from Table 5.1. The robot has lost 3 sonar sensors and is unable to detect any object within the field of those

sensors. The robot is therefore rotated -60° so that the 3 remaining working sonar sensors can detect the object.

Table 5.1: Sonar Sensor Fault Scenarios

'Enabled' sonar sensors positions, used to compensate for fault	**Angle of 'Enabled' sonar sensor on the array**	**'Disabled' sonar position and (angle on the array)**	**Robot rotation(s) required (+ or -)**
Scenario 1 - the sonar sensor at position 3 has become disabled			
2	30°	3 (10°)	-20°
Scenario 2 - the sonar sensor at position 3 and 2 have become disabled			
4 1	-10° 50°	3 (10°) 2 (30°)	+20° -20°
Scenario 3 - the sonar sensors at position 2, 4, 5 and 6 have become disabled			
1,3 3 3	10°, 50° 10° 10°	2 (30°), 4(-10°) 5 (-30°) 6 (-50°)	-20° -40° -60°
Scenario 4 - the sonar sensors at position 4, 5 and 6 have become disabled			
1,2,3	-10°, -30°, -50°	4 (-10°), 5 (-30°), 6 (-50°)	-60°

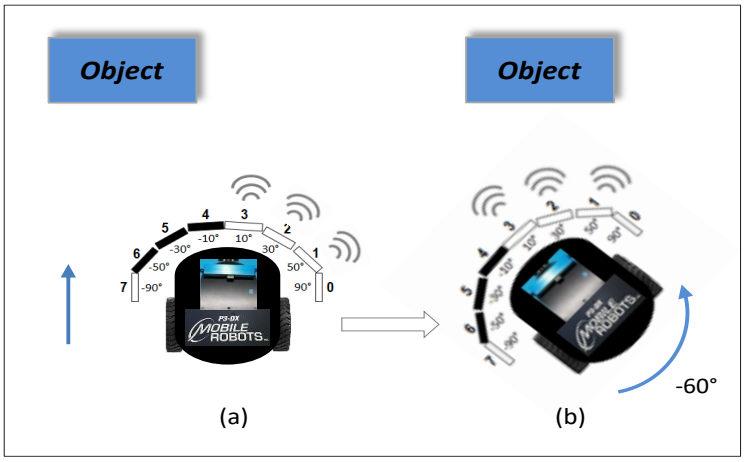


Figure 5.14: (a) Sonar sensors (4-6) are disabled. (b) The P3-DX robot is required to rotate -60° so that the *object* can be detected (using sonar sensors (1-3)).

5.6.1.2 Fault Compensation Algorithms

When the *disabled* sonar sensors are first discovered, the P3-DX robot is stopped, and analysis takes places to evaluate the extent of the fault. If there are multiple

sonar sensor faults, then the number of rotations will increase. Table 5.1 shows the sonar fault scenarios including the number of rotations required to compensate for the disabled sensors. Utilizing the autonomic Monitor and Analysis processes, the 'disabled' sonar sensors are identified. This information is then passed to the Adjustment process. The Adjustment process engages a policy that can utilize the autonomic *self-adjustment* algorithm. The *compensation* policy for disabled sonar sensors is presented in Algorithm 4 and Algorithm 5. **Algorithm 4:** - is used to work out the position of the *disabled* sonar sensors; it then calculates how much rotation is required for the remaining *enabled* sonar sensors to take the place of the *disabled* sonar sensors. **Algorithm 5:** - is used to work out the minimum number of robot *rotations* required to compensate for the faulty sonar sensors. Finally, all rotation values are sent from the (*Adjustment layer [array]*), to the Operation Policy in the System Manager, to execute the physical robot *rotation* commands required to compensate for the faulty sonar sensors.

Figure 5.15 shows how scenario 4 (sonar fault) from Table Table 5.1, can be dealt with using the *compensation algorithms*. The *compensation* strategy allows the robot to detect an object (see Figure 5.15 (a), Sonar 4 reading) and therefore will be able to take action to avoid collision. When Algorithm 5 has been executed, it will return the 'rotation' values required to compensate for the fault.

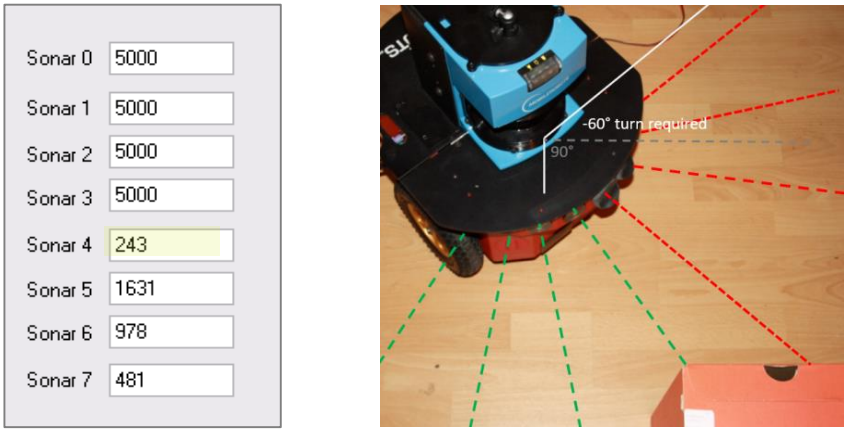


Figure 5.15: Example: when applying the compensation algorithm, Sonar 4 (a) is able to detect the object (b), after a 'rotation' command to the robot has been implemented.

ALGORITHM 4: Compensation For Disabled Sonar Sensors (Part 1)

Input: sonarArray[] = enabled/disabled sonar sensor positions
disabledArray[] = disabled sonar 'angle' position values
enabledArray[] = enabled sonar 'angle' position values
lsa = -50° (lowest sonar sensor angle)
hsa = 50° (highest sonar sensor angle)
ia = 20° (incremental angle value)
av = 0° (angle value initialized for each sonar sensor)

Output: The combinationArray[] = $difference$ value required for an 'enabled' sonar array to take the place of a 'disabled' sonar array.

```
i = 0
for (av = lsa; av < hsa + 1; av = av + ia) do
    if (sonarArray[i] == 'disabled') then
        disabledArray[i] = av;
    end
    if (sonarArray[i] == 'enabled') then
        enabledArray[i] = av;
    end
    i = i + 1
end
ii = 0  (inner index)
oi = 0  (outer index)
av = 0  (reset angle value)
for (dv < disabledArray count) do
    for (av = ia; av < hsa + 1; av = av + ia) do
        if (enabledArray[ii] == (disabledArray[oi] + (-av))) then
            combinationArray[ii] = av;
        end
        if (enabledArray[ii] == (disabledArray[oi] + (av))) then
            combinationArray[ii] = -av;
        end
        ii = ii + 1
    end
    oi = oi + 1
end
```

ALGORITHM 5: Compensation For Disabled Sonar Sensors (Part 2)

```
Input: calcArray[] = the 'sorted' angle values needed for compensation.
       combinationArray[] pre-populated (See Algorithm 3).
Output: rotateArray[] = this array will contain the rotation values the
        robot needs to perform to compensate for the sonar sensor fault
var nearestValue = 0; (find the nearest position value)
var sonarResultCount = combinationArray.Count;
for (int index = 0; index ¡ sonarResultCount; index++) do
    nearest = ca.OrderBy(x => math.abs(long)x − 0)).First();
     combinationArray.Remove(nearestValue);
     calcArray.Add(nearestValue);
end
int eSi = 0; (enabled array index);
foreach ( int calc in calcArray) do
    foreach ( string enabledSonar in enabledArray) do
        if (disabledAr-
         ray.Contains((int32.Parse(enabledArray[eSi].ToString()) +
         (calc)).ToString())))) then
            disabledArray.Remove((Int32.Parse(enabledArray[eSi].ToString())+
             (calc).ToString());
            if (!rotateArray.Contains(calc)) then
                rotateArray.Add(calc);
            end
        end
        eSi++;
    end
    eSi = 0;
end
```

5.6.1.3 Fault Compensation Testing

By introducing a *compensation* policy to handle sonar sensors faults, the Pioneer P3-DX robot can still detect objects. The robot now must make changes in how it executes its tasks. In Figure 5.16, the robot has lost half of the sonar sensors (1-3). However, sonar sensors (4-6) are still functioning. In this example, the robot is instructed to stop every at every 200mm interval. The robot will then rotate (using the compensation policy), so that the sonar sensors that are still functioning, will be able to detect any objects within the robot's path. Although the *compensation* policy is effective in detecting objects using limited sensor ability, there is a impact on executing the task set out for the robot. The time to complete the task will

increase as the robot has to stop and check for objects. There is also an increase in demand for more power resources, as the robot will be taking longer to complete the task and executing multiple 'movement' commands in order to detect objects along the robot's path.

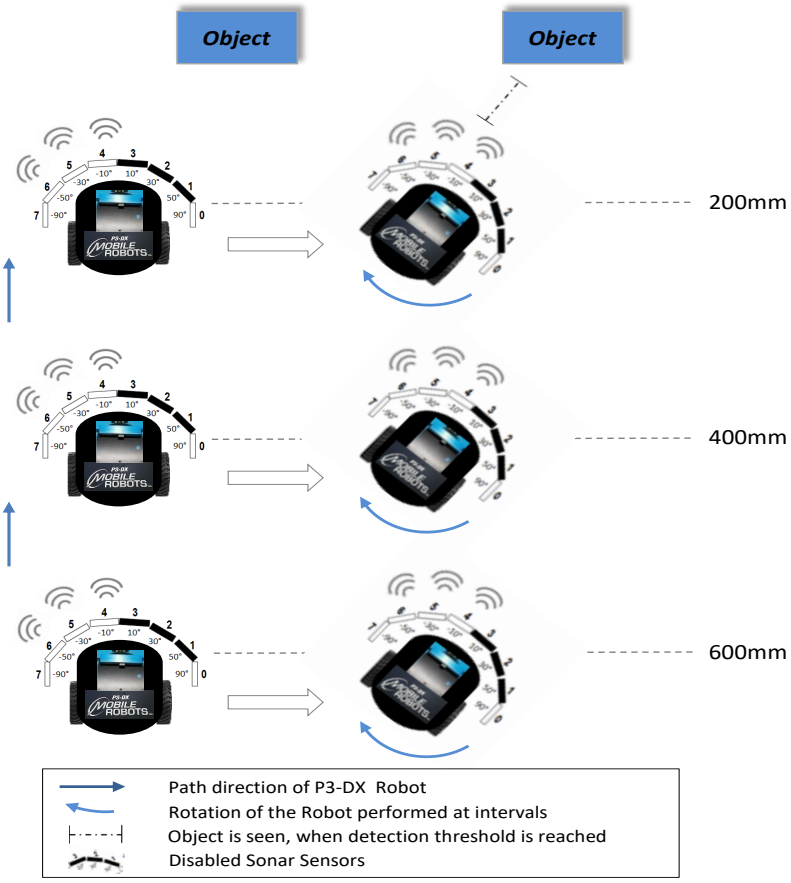


Figure 5.16: When a sonar fault is detected, the Robot is stopped at selected intervals. The robot is then rotated to check for possible objects.

As with Chapter 4 (Wheel Alignment Fault Handling), the importance of historical data is key to understanding fault scenarios. Comparing current data with data recorded from previous tasks, helps us to establish if a fault has occurred and what impact it has on the performance of the mobile robot. Using this *knowledge* we can then make calculations for compensating for the fault. We also discovered in this case study that setting fixed *threshold* values for detecting objects could not always be applied. The sonar sensor is octadecagon in profile, therefore each sensor can be at a slightly different distance from an object. For example: if sensor A is

nearer to an object than sensor B, then the distance values they report back will be different. If we use a +- 10mm threshold value to detect an object for sensor A, then sensor B's distance calculation would lie out of that threshold and could wrongly be flagged as a possible faulty sensor during the Awareness process. This is described as a threshold being 'over sensitive'; we therefore have to allow for the physical difference between two adjacent sensors before applying a threshold value (see Figure 5.10).

5.7 Evaluation

Using the AIFH architecture a *Reactive* and *Proactive* control loop was integrated into the system using an Autonomic Manager. The Autonomic Manager worked alongside the System Manager, so that the robot tasks are constantly monitored. The Autonomic Manager consisted of three layers - Awareness, Analysis and Adjustment. *Self-awareness* has the ability to detect and report on component faults within a robot. *Self-awareness* is more than just monitoring the system data. *Self-awareness* can interpret the data and inform Users of possible impending faults rather than just reacting to a 'alarm' type fault. *Self-analysis* is key to determine the extent of a fault. This information is crucial to determine if the fault can be compensated for. *Self-adjustment* allows the robot to continue to function even when operating with a faulty component.

However, the more sonar sensors that are at fault, then the greater impact it will have on the operational efficiency of the robot. Figure 5.17 shows a selection of possible fault combinations (odd numbered fault scenarios), of 31 in total, from a possible 62 (as discussed in Section 5.6.1.1). The greater number of 'disabled' sonar sensors on the array, then the greater number of robot rotations are required to compensate for the faulty sensors. This ultimately will have an impact on task time and power required. When a fault occurs in the sonar sensor array, then the mobile robot *self-adapts* to the changes in its sensor state.

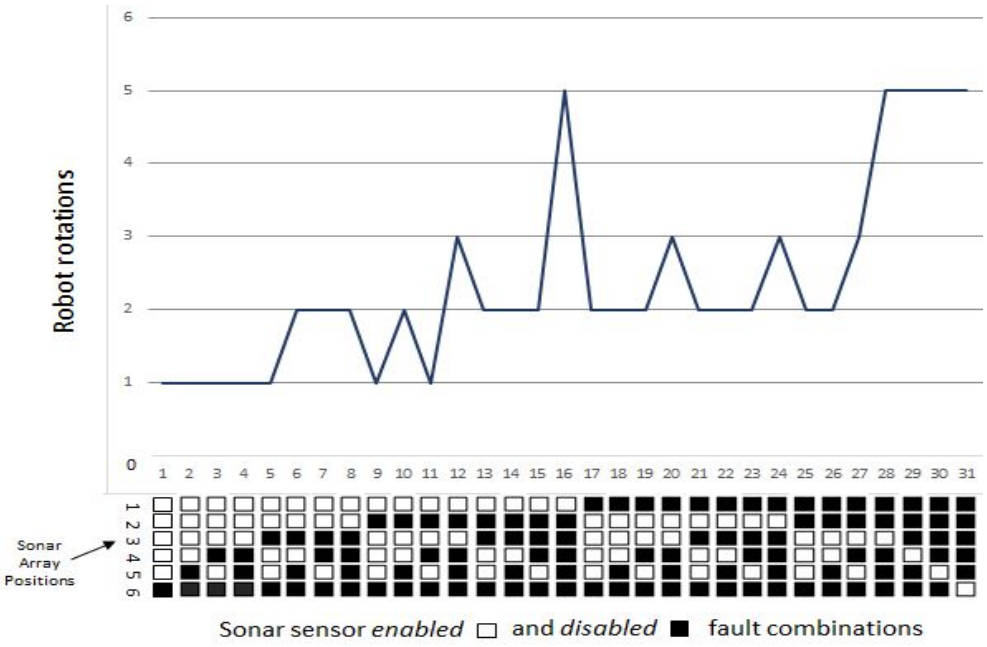


Figure 5.17: The increased in the number of sonar sensor faults will also increase the number of rotations required to compensate for the fault.

5.8 Summary

Autonomic Sonar Sensor Fault Management for Mobile Robots case study set out to demonstrate how an Autonomic Management System can be implemented to handle sonar sensor hardware failures in a mobile robot such as the Pioneer P3-DX. The System Development Life Cycle (SDLC) was implemented as a Research Model to describe a sequence of activities including requirements, design, implementation testing and evaluation. The Research question posed for this case study stated - if the AIFH architecture (derived from the MAPE-K and IMD autonomic models), can be applied to detecting sonar sensor faults on a mobile robot and if possible, compensate for those faults? Can the mobile robot adapt to its changed environment and continue to function even with a sonar sensor fault?

The AIFH Knowledge Base is used to gather 'task' data from the mobile robot and to provide 'tolerance' data for sonar sensor information. Using the Reactive Control feedback loop, the *Awareness Layer* can report directly to *Analysis Layer* if any of the sonar sensors are displaying a *disabled* state. Using the Proactive Control feedback loop, the *Awareness Layer* is also capable of detecting unusual

readings between neighbouring sonar sensors. Those sensors under investigation are then passed to the *Analysis Layer* for further processing. The AIFH Knowledge Base contains a policy to allow the *Analysis Layer* to determine if there is any faults with those sensors under investigation. If a fault is detected, then this information is passed to the *Adjustment Layer*. In the *Adjustment Layer*, two policies from the Knowledge Base are required to calculate how the remaining *enabled* sonar sensors can be implemented to compensate for the *disabled* sonar sensors; a further policy is then used to send 'movement' commands (rotation, forward motion) to the System Manager.

This case study shows how autonomic principles can be employed so that a mobile robot can *self-adapt* to changes to its sonar sensor functionality. Not only can faults be identified but also a compensation strategy can allow the robot to continue to function. However, laboratory experiments showed that as the number of disabled sonar sensors increased, then the time for the robot to complete its task is also increased. This can also have an impact on the power resources available to the robot. Research carried out in this case study provides valuable data for generating a Generic Autonomic Architecture for handling component faults in a mobile robot (see Chapter 7).

Chapter 6

Autonomic Management for Mobile Robot Battery Degradation - case study

6.1 Introduction

This case study is concerned with how the Autonomic principles (discussed on Chapters 2 and 3) can be applied to managing mobile robot power resources when experiencing battery degradation.

Lead-acid batteries which contain a lead-calcium grid structure, are vulnerable to an aging process due to the fact they are repeatedly cycled [104]. This aging process is known as battery degradation. Various models for battery degradation have been investigated in [105], to determine the *degradation curve* for any lead-acid battery. Data provided from *degradation curve* can assist researchers in predicting when battery degradation will affect the system operations. With the vast improvements in computer technology, the need for improved battery design and a reduction in battery degradation, are in demand more than ever [106]. In research conducted in [107], the Smart Battery System (SBS) offers the ability to adjust the charging profile in response to actual requirements i.e., charging voltage and charging current. SBS can also monitor various charge states and raise Alarms if a damaged battery is detected. Although SBS is aware of charge states, it has no formal processes

to deal with battery degradation. Prognostic-enabled Decision Making (PDM), is a research area that aims to integrate prognostic health data and knowledge of future operations, into the decision making when selecting a particular action within the System [108]. Health checks include reporting battery capacity faults and other operations including heavy load capacity and temperature variations. PDM is employed as a fault reporting tool rather than a tool that can react to identifying battery degradation.

This case study employs a simulated battery configuration based on the actual lead-acid battery contained in our laboratory robot (Pioneer P3-DX). The research will concentrate on how battery degradation can affect how a mobile robot performs basic tasks, like moving from one location to another. Task management is important especially if the robot is operating on remote environments. Investigations will also focus on the importance of how the battery charging is maintained. Research in this case study will also focus on how to compensate for battery degradation so that the mobile robot can still complete its allocated tasks. Although some considerable research has been carried out in applying autonomic principles with fault detection in robots, there is very little evidence of any research carried out in autonomic power management in mobile robots. Robot sensors and effectors are vitally important in the operation of mobile robots, but they rely heavily on the power supply within the robot. If power supply is degraded or compromised, then this will have a detrimental effect on the performance of sensors and effectors.

In Chapter 5, the AIFH architecture was further developed by using Knowledge Base attributes (within the Autonomic Manager), to handle sonar sensor faults in mobile robots. However, the use of both *reactive* and *proactive* feedback loops where limited. In this case study, the *proactive* loop takes an important role in alerting the User/Mission Control of an impending fault. This describes the AIFH having *self-awareness* and the ability to make decisions when faced with a changing environment.

This case study is organized as follows: Section 6.1.1 describes how the Research Methods is implemented by utilizing the SDLC (Software Development Life

Cycle) model. Section 6.2 presents Conceptual Requirements which describe the Research question, goals and resources needed to complete the case study. Section 6.3 presents the Conceptual Design and describes how the Autonomic Management System has been adapted to use the AIFH architectural model to handle mobile robot power management faults. Section 6.4 presents Implementation of the battery degradation fault scenario and how Awareness, Analysis and Adjustment have been used to evaluate and compensate for a fault. Section 6.5 demonstrates through testing, how the *compensation* algorithm for the battery degradation fault performs. Results from the testing are then analysed. Section 6.6 presents an evaluation of the case study. Section 6.7 concludes the case study with a summary statement.

6.1.1 Research Method

The SDLC model approaches solving a given problem in well-defined steps. Figure 6.1 shows how the SDLC Model can be applied to the Autonomic Robot Battery Degradation Fault case study.

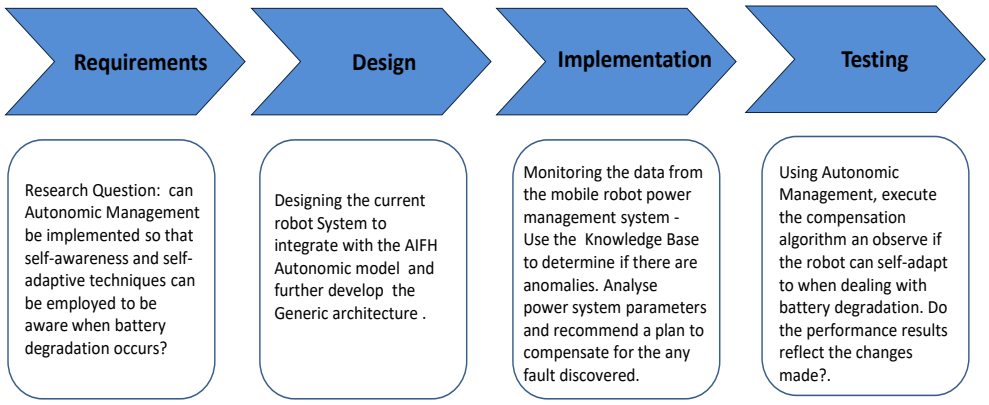


Figure 6.1: SDLC Model used in the research methodology for Autonomic Wheel Alignment

For the purpose of this case study, the SDLC model from Figure 6.1 is employed. SDLC model ensures that all the work carried out within the case study is documented and results generated.

- **Requirements** - lays out the broad research objectives of the particular case study. It is a detailed investigation of the system and is carried out in ac-

cordance to the objectives proposed. It involves a detailed study of various operations performed by the system and their relationships within and outside the system [92]. To develop the research question successfully, a simulated robot battery environment is required to emulate the characteristics of the Pioneer P3-DX robot.

- **Design** - based on the information collected at the requirements stage. The logical system design is arrived at as a result of system analysis and how it is converted into physical system design. The SDLC process moves from the **what** in the requirements phase to the **how** in the design phase [92]. For this case study, the design is concerned with applying the Autonomic Model to handle how the power management system in the robot can adapt to the discovery of a battery fault. In Chapter 3, Section 3.2.3, the initial concept of the AIFH architecture was introduced. In this case study, the findings accumulated in the *autonomic battery degradation* research, will provide a means to further develop the AIFH architectural model. The layers contained in the AIFH architecture are *Awareness*, *Analysis* and *Adjustment*. *Self-Awareness*, is the most important, as it is an indicator to the mobile robot autonomic manager that there is a fault within the system. *Self-Analysis* is used to evaluate the extent of the fault. *Self-Adjustment* takes the analysed data and applies a specialized algorithm to compensate for the fault.

- **Implementation** - the system design needs to be implemented to make it a workable system. To create the battery degradation experiment, a simulation of the P3-DX robot is employed. The properties from the 'actual' P3-DX robot battery are used as parameter values in the experiment. Using the P3-DX robot DOD (Depth of Discharge) chart, then the battery degradation process can be simulated within the programming. In this case study, programming is implemented to allow the Autonomic Manager access to the power management data in the AIFH Knowledge Base. Further programming is implemented to evaluate any fault data that supplied by the *Monitoring* process. The *Analysis* process is used to investigate the extent of the fault. When the

fault has been analysed, then additional programming is required for formulate a possible compensation strategy.

- **Testing/Evaluation** - involves system integration and system testing of the programs and procedures coded at the implementation phase. Testing is a method of testing the system against the requirements and design. For this case study, testing will involve evaluating the how the simulated P3-DX battery performs when the battery is in optimal condition and when the battery is exposed to degradation.

  Test Strategy:

  1. Using the simulation program for the P3-DX robot, run a robot task over a fixed distance, including speed and power requirement in watts (W).
  2. Set the battery cycle value of the robot to represent the battery at optimal charge capacity.
  3. Use the battery properties (taken from the actual P3-DX battery), to include parameters such as Watt Hours (WH), battery rating (Ah) and voltage (V).
  4. Record the amount of battery charge needed to complete the task.
  5. Is there enough battery change to complete the robot task?
  6. Repeat the simulated task for the mobile robot with the same parameter values.
  7. Set the battery cycle value to represent the battery when it is near its 'end of life' state.
  8. Record the amount of battery charge needed to complete the task.
  9. Is there enough battery change to complete the robot task?
  10. If there is not enough battery charge to complete the task, then declare a fault.
  11. How did the battery degradation *compensation* algorithm perform, when applied to the simulated robot task?

6.2 Conceptual Requirements

The Requirements phase in a SDLC model is the most crucial step in creating a successful case study. Requirements define the problem, objectives and the resources needed to complete the study.

6.2.1 Research Question

6.2.1.1 Goals

Using AIFH model (see Fig. 3.2), as a baseline, the AIFH architecture is further developed in this case study to establish if an autonomic architecture can be used to detect and compensate for robot power management faults.

- **Awareness** - can processing past and present experimental data, allow the Autonomic Manager the ability to decide if there is a power fault on the robot?
- **Awareness** - can processing past and present experimental data highlight any trends that the capacity of the robot is reduced as the battery cycle count increases?
- **Analysis** - can analysis provide the means to establish the extent of the fault?
- **Adjustment** - can an *adjustment* strategy be provided that will compensate for the battery degradation?

6.2.1.2 The Experiment

A robot simulation program (see Figure 6.3), is used to render the Pioneer P3-DX robot. The battery values are taken from the actual P3-DX battery properties (see Section 6.2.2.3). The first part of the experiment shows how the battery performs using a chosen DOD (Depth of Discharge) strategy (see Section 6.2.4). The second part of the experiment simulates a series of robot tasks. The battery properties and task instructions supply the parameter values used in each robot task. These values are then integrated into equations to establish battery capacity required for each task (see Section 6.4.1). Depending on the 'charge' cycle used, then battery degradation will start in influence the task result. Finally, if battery degradation has been flagged

by the Autonomic Manager, then a *compensation* strategy is employed within the task to handle the *battery degradation* fault (see Section 6.5.1).

6.2.2 Resources required

6.2.2.1 Battery Degradation in Lead-Acid Batteries

Battery degradation is unavoidable in lead-acid batteries; however, the rate of degradation can be predicted depending on how the battery is managed during its lifetime. However, the rate of degradation can be managed depending on some known factors [104].

6.2.2.2 Battery Degradation Factors

- Loss of active material from positive plates.
- Loss of capacity due to the physical changes in the active material of positive plates.
- Temperature elevated temperatures reduce battery life.
- Cycle service discharge cycles reduce battery life.

6.2.2.3 Pioneer P3-DX Robot - battery properties

Experimentation in this case study with real batteries is unrealistic, as battery degradation can take many months to occur. However, to accurately simulate the battery performance, the properties of the actual Pioneer P3-DX robot battery is implemented. The Pioneer P3-DX uses the YUASA NP Series (NP7.5) battery. The NP7.5 data sheet provided all the necessary battery *capacity* and *charge* ratings [109].

The life of a battery can be described as the number of 'charge' cycles it can produce before being discarded. The number of charge cycles available greatly depends on how the battery is charged/discharged during its lifetime [110]. DOD (Depth of Discharge) is used to describe how deeply a battery is discharged. The less a battery is discharged then the greater the number of 'charge' cycles you will

get from the battery over its lifetime. Fig. 6.2 shows the DOD characteristics of the lead-acid battery used in the Pioneer P3-DX robot. [16].

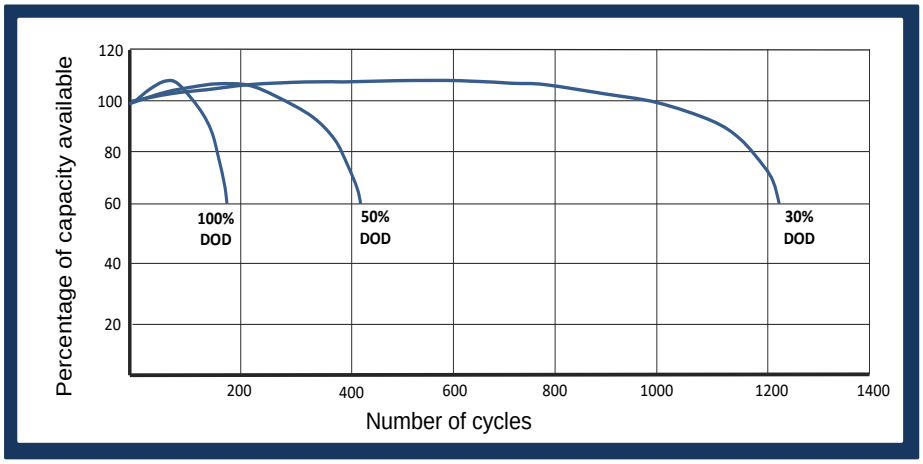


Figure 6.2: The DOD (Depth of Discharge) characteristics for the lead-acid battery used in the Pioneer P3-DX robot [16].

If a battery is discharged at nearly 100 %, then the number of battery cycles available during the its lifetime is greatly reduced. If the battery is only discharged to 30 %, (see Fig. 6.2), then numbers of charge cycles available will increase. Battery degradation can be managed by applying basic principles when charging the battery.

- A lead-acid battery should never be discharged below 80 %, otherwise over time the battery will be damaged [111].
- To prevent sulfating and stratification, a lead-acid battery needs an Equalizing Charge. The battery is charged at a higher voltage and this is required every 10 charge cycles [111].
- The Pioneer Robot battery supply should be maintained above 11 VDC (Voltage Direct Current). If it falls below 10 VDC, the battery warning signals.

6.2.3 Simulated Battery Performance

Experimentation with real batteries in a mobile robot is unrealistic as battery degradation can take many months to occur. For this case study, the Pioneer P3-DX robot

battery is simulated using MRDS (Microsoft Robotics Developer Studio) and SPL (Simulation Programming Language) editor to provide a graphics environment (see Figure 6.3).

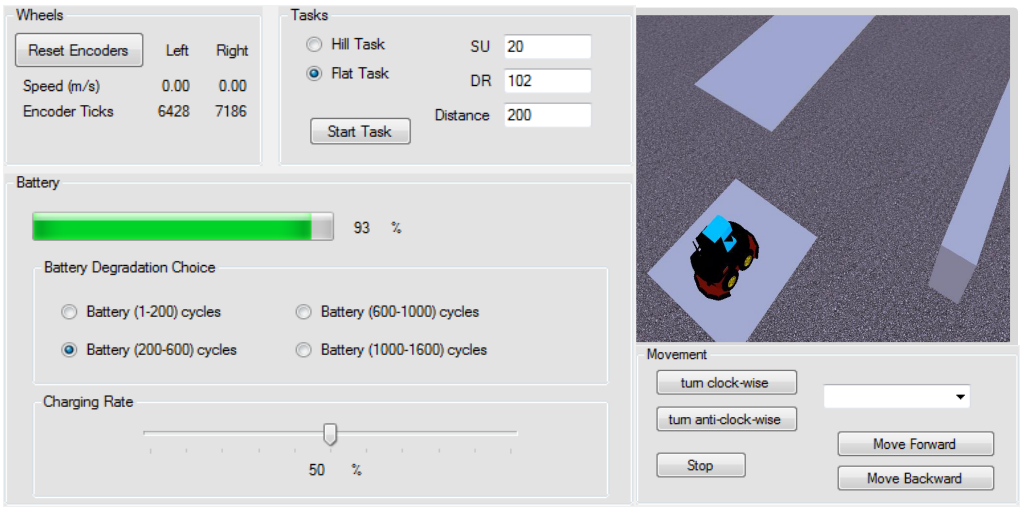


Figure 6.3: Battery simulation program (developed by the author), using (MRDS) and robot (P3-DX) rendering using (SPL).

6.2.4 Simulated Battery setup task

In the simulated *Setup Task*, the mobile robot travels from point A to point B, which is measured at 200 meters. Table 1 shows the DOD rates at 100%, 50% and 30%. The cycle number represents the number of times the battery has been re-charged.

The following parameters are defined to create a basic battery simulation involving the DOD values (see Table 6.1).

DR depth of charge rate (this is the percentage of dis-charge used before the battery is cycled). **DOD** depth of discharge (this is the percentage available in the battery to discharge). **DP** discharge percentage (this is the amount of discharge available at a particular cycle count). **BP** battery percentage (this is the amount of battery charge required to complete a task). **SU** single unit (this unit is calibrated 1% of battery power and will move the robot a distance of 20 meters). **DISTU** distance unit (this is the number of SUs required given a distance value).

Table 6.1: Shows the percentage rate (DR) and depth of discharge (DOD) for the P3-DX battery. When DOD falls below 60%, then the battery loses its ability to hold a significant charge and therefore DOD is denoted as '-'

No.	Cycle Number	DOD charge @ 100 %	DOD charge @ 50%	DOD charge @ 30%
1	0	100	100	100
2	20	104	101	101
3	40	108	103	101
4	80	110	104	102
5	100	101	107	103
6	120	95	107	103
7	140	91	108	103
8	160	80	108	104
9	180	60	109	105
10	200	-	110	106
11	240	-	103	106
12	280	-	105	107
13	320	-	102	107
14	340	-	98	108
15	360	-	95	108
16	380	-	91	109
17	400	-	86	110
18	500	-	80	110
19	600	-	60	110
20	700	-	-	108
21	800	-	-	106
22	900	-	-	102
23	1000	-	-	98
24	1050	-	-	95
25	1100	-	-	90
26	1150	-	-	80
27	1200	-	-	60

Equation (6.1) is used to calculate the **DISTU** value.

$$DISTU = \frac{Distance}{SU} \tag{6.1}$$

Using the Cycle Number value from Table 6.1, the **DP** can be calculated using Equation (6.2)

$$DP = \frac{DOD * DR}{100} \tag{6.2}$$

The percentage of charge required to complete a task is calculated using Equation (6.3). This percentage is important, as the mobile robot system manager needs to establish if the there is enough charge left in the battery to complete the task.

$$BP = \frac{100 * DISTU}{DP} \tag{6.3}$$

Figure 6.4 shows the percentage charge required to complete the task when the battery is at various stages within its cycle. In this example the DOD rate is set at 50%. The robot task distance = 200 meters. In the early stages of battery life, the battery charge requirements for the task remain constant. However, after 240 charge cycles, the percentage of charge increases as the battery degradation increases. This would have an impact on tasks that required the robot to travel long distances.

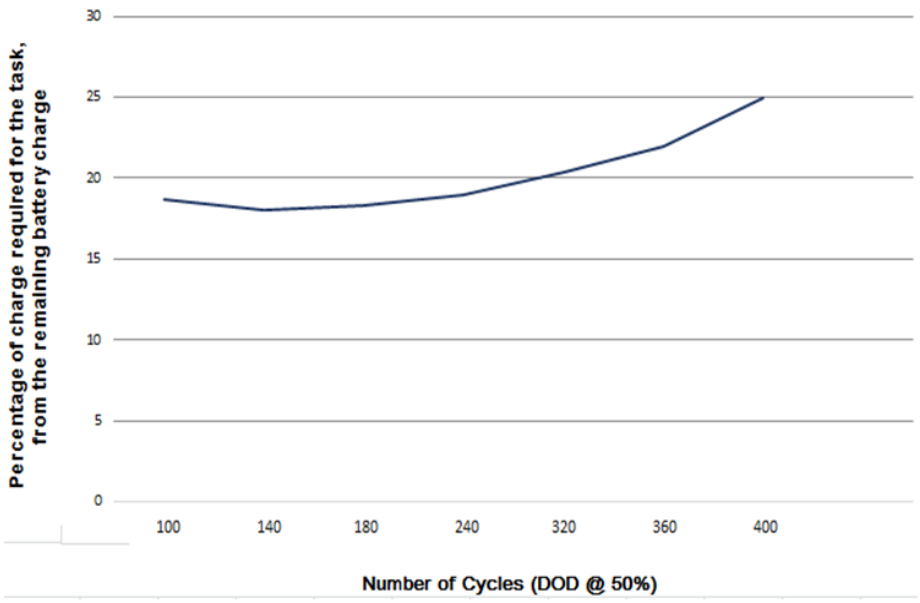


Figure 6.4: Shows the percentage charge required to complete a task when the battery is at various stages within its cycle - using 50 % DOD.

6.3 Conceptual Design

6.3.1 Autonomic Battery Management

In Chapter 4 and Chapter 5, the AIFH architecture was developed to handle both *wheel alignment* faults and *sonar sensor* faults. In this Chapter the AIFH model is further developed to handle power management fault (battery degradation). The AIFH model contains a System Manger and an Autonomic Manager. The System Manager is responsible for initiating the 'control' loop - Reactive and Proactive. The main feedback loop (Reactive), will traverse through each of the Autonomic Manager layers - *Awareness*, *Analysis* and *Adjustment*. The secondary feedback loop (Proactive), will operate in the *Awareness Layer*. The AIFH architectural model for the robot Battery Degradation fault handling is shown in Figure 6.5.

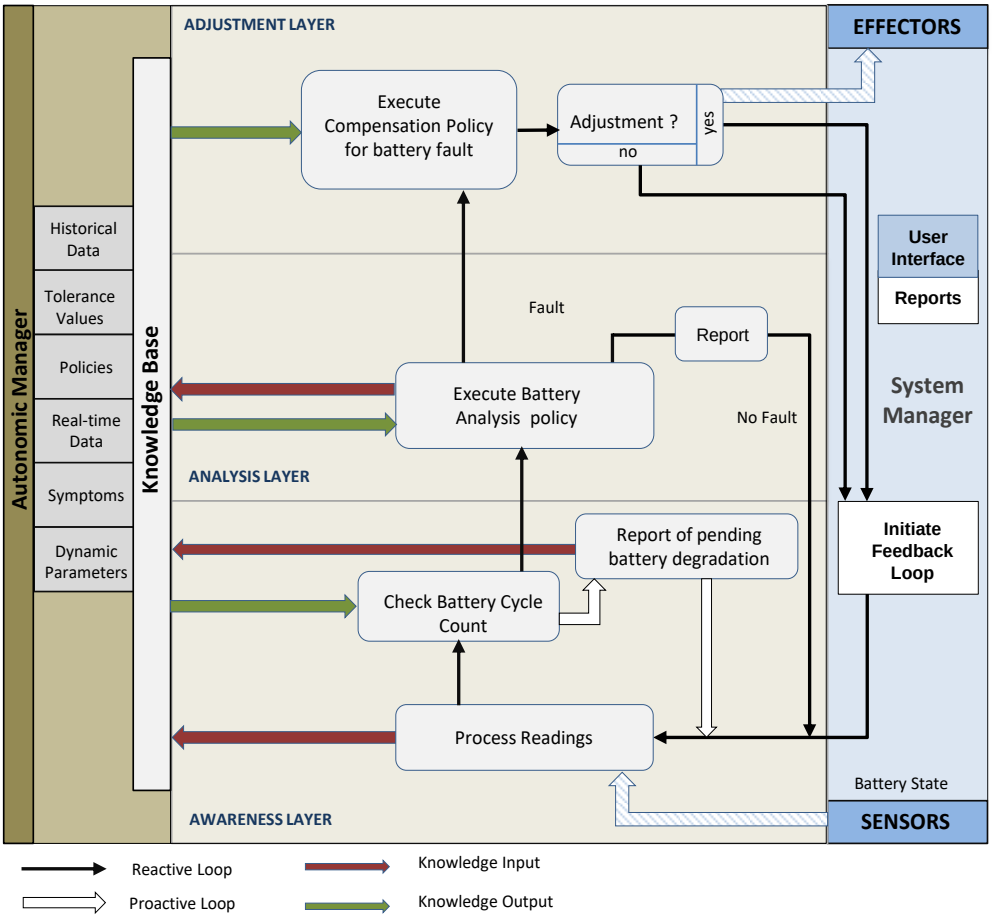


Figure 6.5: Autonomic Model for Battery Degradation Management

1. *Awareness Layer* - this layer is responsible for flagging a potential fault within the power management system on the robot. The Awareness layer will use the Knowledge Base data to establish if there is potentially a fault. The Reactive Control loop will initiate the 'Process Readings' module. This module will update the Knowledge Base (Dynamic Parameters), with the current state of the battery i.e., current charge available and current cycle count. The Reactive Control loop will then execute the 'Check Battery Cycle Count' module. Using the Knowledge Base 'historical data' and dynamic parameters', this module can make a decision as to whether there is a potential battery degradation fault. If a possible fault has been identified, then all the collected data is passed to the Awareness Layer. The Proactive Control loop uses the Knowledge Base 'historical', 'real-time' and 'dynamic parameters' data to establish if there is a possible pending battery degradation fault. The Proactive Control loop has access to the DOD (Depth of Charge) value and can match this with the current battery 'cycle count'. If a possible battery degradation fault is imminent, a report can be sent to the System Manager User Interface. In Figure 6.6, the chart shows the *pre-degradation* phase; the Proactive Control alerts the System Manager during this period of the battery lifeline before the *degradation* phase occurs.

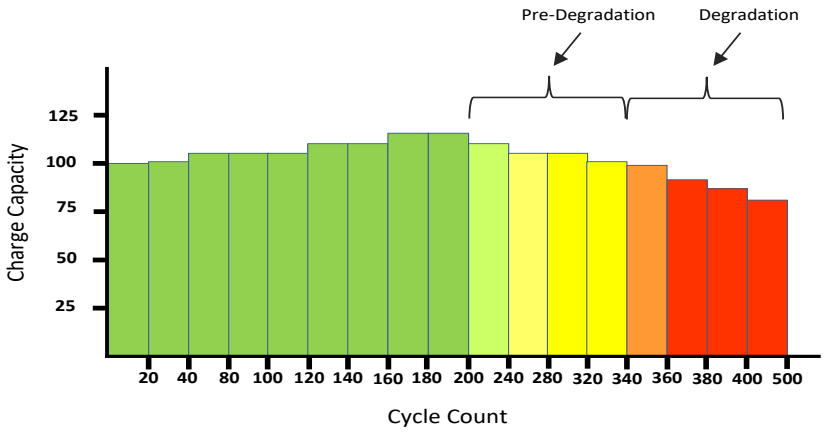


Figure 6.6: The cycle lifetime of the P3-DX battery using a DOD of 50 %. The Proactive Control loop is concerned with the pre-degradation phase.

2. *Analysis Layer* - this layer uses data received from Awareness layer to establish if a there is a fault in the power system within the robot. The Analysis layer uses a 'Battery Analysis' policy from the Knowledge Base to check on the state of the battery. The 'Battery Analysis' policy uses the Knowledge Base 'tolerance values' and 'dynamic parameters' together with 'real-time' task data, to establish if there is a battery fault. If the tolerance threshold limits are exceeded, then battery is now declared as being in a 'degradation' state. However, if the battery is operating within tolerance limits, then the Reactive Control loop will initiate the 'Report' module to inform the System Manager that 'no fault' was found - see Figure 6.5.

3. *Adjustment Layer* - this layer receives data from the Analysis layer containing information about the battery degradation fault. This layer is used to make possible adjustments to the robot System Operations in order to compensate for the fault. The Reactive Control loop initiates the 'Execute Compensation Policy for battery fault' module which utilizes the Knowledge Base 'policies'. This policy can then be used to make adjustments i.e., reduce the speed of the motors and therefore, reduce the amount of battery power needed to complete a task. If an *adjustment* can be made, then the Adjustment Layer will pass the necessary data to the System Manager to make the adjustment. However, if no possible adjustment can be made, then the Reactive Control loop will send a report to the System Manager - see Figure 6.5.

The Autonomic Manager will periodically monitor the data supplied by the robot sensors and battery monitor. If performance data is within acceptable thresholds, the Autonomic Manager will not intervene. The Autonomic Manager will also update historical as each task is completed. If similar task is requiring more battery capacity that it did previously, then this could indicate that the battery is coming to the end of its life in terms of a useful power source. If a battery cannot retain a charge of more than 80 %, then it needs to be replaced [104]. Alternatively, the Autonomic Manager could enforce a policy where the DOD rate is changed. Lowering the DOD rate (at the charging station) can extend the batteries life cycle count.

6.4 Implementation

6.4.1 Autonomic Battery Power Management

The Pioneer P3-DX robot contains components that require a certain level of power-input. The lead-acid batteries contained within the robot supply the necessary power for the components. Research conducted in [17], shows the relative power required for each of the Pioneer P3-DX components - see Table 6.2.

Table 6.2: Power requirements for each component in the Pioneer P3-DX robot [17].

Component	Power	Percentage
Motion	2.8W ~ 10.6W	12% ~ 44.6%
Sensing (sonar)	0.58W ~ 0.82W	1.9% ~ 5.1%
Micro-controller	4.6W	14.8% ~ 28.8%
Embedded Computer	8W ~ 15W	33.3% ~ 65.3%

For this experiment in the case study, investigations where carried out in how battery degradation can affect how much power is available for the 'Motion' component during different stages of the lifetime of the battery. Fig. 6.7 shows how the research conducted in [17], describes the amount of power needed for the robots 'Motion' component when driven at various speeds.

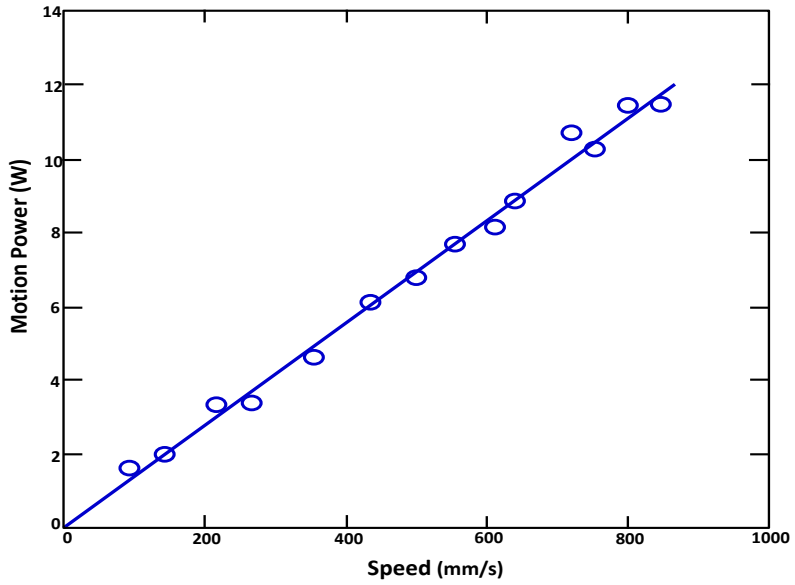


Figure 6.7: shows the power(W) required for the 'motion' component in the P3-DX when driven at various speeds [17].

The autonomic battery power management design for the mobile robot includes a System Manager and an Autonomic Manager - see Figure 6.5. The System Manager accepts input from the User Interface and translates this into *commands* which will provide direction and speed for the Pioneer P3-DX robot. The Autonomic Manager monitors and analyses tasks performed by the robot. The Autonomic Manager considers the current battery 'cycle' value and the current power (W) utilized by the robot for the 'Motion' component. If the threshold limits of the battery in its present state are being exceeded, then the Autonomic Manager will make the necessary adjustments to the power (W) level that is provided to the 'Motion' component.

6.4.1.1 Robot Task One - Motion Management

Using data collected by research conducted in [17], battery data from Fig. 6.2 and the User input values, we can construct *parameter* and *test* values - see Table 6.3.

Table 6.3: Robot Task One: setup values for robot running @ battery cycle 0.

Parameters	Values
RS Robot Speed (mm/s)	600 (0.6 M)
PR Power required (W)	8
DIS Distance to travel (M)	5,000
T Time (hr.)	2.31
WHU Watt Hours used	18.48
WC Watt Hour capacity	25.2
BC Battery Cycle	0

The battery in the Pioneer P3-DX provides 84 watt-hours of power capacity [112]. When the battery is at cycle 0, then the battery capacity is 100%. The battery rating is 7000mAh offering 12 volts [16]. The following equation (6.4), is used to calculate the *watt hour* value for the battery. (E = Energy, Q = milliamp hours and V = Voltage). This will give 84 watt-hours as described in [112].

$$E(wh) = Q(mAh)\ x\ \frac{V(v)}{1000} \tag{6.4}$$

Using the values in Table 6.3, we can establish the value of the 'motion' component in terms of *watt hours* used.

$$WHU = PR\,x\,T \tag{6.5}$$

To prolong the life of the battery, a DOD rate of 30% is employed - see Fig. 6.2. When adopting a 30 % DOD rate, this means that the battery is never allowed to fall below 70% charge capacity. The battery at 100% charge gives 84 watt-hours of power, however, if 30% DOD rate is used, then only 25.2 watt-hours available for the Pioneer robot at battery cycle 0 (see equation (6.6) for how watt-hour capacity **WC** is calculated at the DOD rate). Using the **WHU** value from equation (6.5), we can then calculate the percentage capacity (**PC**) required for the robot to complete Robot Task (see Table 6.3).

$$WC = \frac{E(wh)}{100}\,x\,DOD \tag{6.6}$$

$$PC = \frac{WHU}{WC}\,x\,100 \tag{6.7}$$

The **PC** value is calculated using equation (6.7). For this experiment, the acceptable *threshold* (**AT**) value for how much capacity a robot task uses, is set to 80%. If the **PC** value is below the **AT** value, then the task can complete successfully. If the **PC** value is above the **AT** threshold value, then task is under threat as it is using full power resource from the battery at the present DOD rate.

The initial robot Task (see Table 6.3), requires a battery charge capacity **PC** of 73.33%, when employing equation (6.7). The **PC** value of 73.33% is below the threshold value (80%) and therefore no adjustment from the Autonomic Manager is required.

6.4.1.2 Robot Task Two - Motion Management (with battery degradation)

In this experiment, the same parameters are used from Robot Task one but in this case, the battery is now in a degradation phase and therefore the amount of charge capacity is reduced. The robot task is run using cycle 1100 (see Table 6.4), which results in capacity dropping from 100% to 90% (see Fig. 6.5). We need to re-calculate the (Ewh) value using equation (6.4). This results in battery capacity being reduced from 84 watt-hours to 75.6 watt-hours. Using the DOD rate of 30%, we now have 22.68 watt-hours available for the task. If we apply equation (6.7), then the task will require 81.48% of battery capacity **PC**, which is above the acceptable threshold **AT** value of 80%.

Table 6.4: Robot Task Two: setup values for robot running @ battery cycle 1100.

Parameters	Values
RS Robot Speed (mm/s)	600 (0.6 M)
PR Power required (W)	8
DIS Distance to travel (M)	5,000
T Time (hr.)	2.31
WHU Watt Hours used	18.48
WC Watt Hour capacity	22.68
BC Battery Cycle	1100

Within the Analysis Layer, a battery degradation fault is identified using the 'Battery Analysis' policy. Algorithm 6 (Battery Analysis Policy), shows how task input and analysis performed by the Autonomic Manager can establish if the **PC** value is within tolerance values. Using the current 'Battery Cycle Charge' percentage and the current 'Battery Cycle Count', a check is made to see if the battery has reached its final 'cycle count' according to its DOD status. If this check returns 'true' then the battery has expired and no compensation for the fault can be applied. If the battery is still functioning, then the equations (6.5), (6.6) and (6.7) are applied in order to establish that the **PC** Percentage Charge is below the 'charge' threshold limit (held in the Knowledge Base - in Tolerance values). If **PC** is above the 'charge' threshold limit, then a fault is declared, and the relevant data is

then passed to the Adjustment Layer to apply a compensation policy - (see Section 6.5.1).

ALGORITHM 6: Battery Analysis Policy - checks that **PC** is within tolerance range.

Input: DOD = *selectChargeRatingForBattery()*
batteryCycleCount = *getCurrentBatteryCycleCount()*
batteryCyclePercentageValue = *getCurrentBatteryCyclePercentage(batteryCycleCount, DOD)*
upperCycleValue = *getUpperCycleValue(DOD).*

Output: thresholdExceeded = if this value is set to true, then the $adjustmentBatteryCompensation()$ algorithm needs to be engaged.

if *(batteryCycleCount* $>$ *upperCycleValue)* **then**
  $batteryExpired$ = true;
end
if *(batteryExpired* $= false$*)* **then**
  double $RS = robotSpeedInput()$;
  int $PR = powerRequiredInput()$;
  double $DIS = distanceToTravelInput()$;
  //Calculate the travel time for task;
  double $T = DIS/(RS/1000)$;
  //Calculate the watt hours used for task;
  double $WHU = T * PR$;
  //Energy available from battery at cycle count position;
  double $Q(mAh) = 7ah * batteryCyclePercentageValue$;
  double $E = Q(mAh) * voltage/1000$;
  //Use the DOD rate, calculate the working battery capacity;
  double $WA = DOD * E/100$;
  //Calculate the percentage capacity required by the robot task;
  double $PC = WHA/WA * 100$;
  //Calculate the percentage of battery capacity required for task does not exceed threshold value;
  if *(PC* $>$ 80%*)* **then**
    $thresholdExceeded = true$;
  end
  //If threshold value is exceeded then call the Battery Adjustment Algorithm;
  if *(thresholdExceeded* $= true$*)* **then**
    $adjustmentBatteryCompensation()$;
  end
end

6.5 Demonstration (testing)

6.5.1 Robot Task Three - Motion Management (with battery degradation) - applying a compensation policy

The Implementation Section (6.4), showed how the robot performed a task with an optimal battery and a battery that was experiencing degradation issues. The experiment in Section 6.4.1.2 showed that the battery degradation led to threshold values being exceeded. To compensate for the battery degradation fault, an *adjustment* policy is required, so that the robot can complete a task with the threshold limits. To bring the task performed by the robot at cycle 1100 below the battery usage threshold value of 80%, we need to reduce the speed and power of the robot. If we use the adjusted values from Table 6.5, the **WHU** value is now at 18.00 using equation (6.5). We can then calculate the **PC** value using equation (6.7). The resulting **PC** value of 79.66% is now below the threshold **AT** value of 80% and therefore the robot can safely complete the task.

Table 6.5: Robot Task Three: compensation - reduce speed @ battery cycle 1100.

Parameters	Values
RS Robot Speed (mm/s)	500 (0.5 M)
PR Power required (W)	6.5
DIS Distance to travel (M)	5,000
T Time (hr.)	2.77
WHU Watt Hours used	18.00
WC Watt Hour capacity	22.68
BC Battery Cycle	1100

6.5.2 Battery Degradation Compensation Algorithm

If the **PC** value is above the acceptable threshold limit, then Algorithm 7 is initiated. This Policy is stored in the Knowledge Base of the AIFH Autonomic Manager. The Adjustment Layer will use this policy to make adjustments to the power requirements of the robot so that less battery charge is required. The Compensation Algorithm 7, will be periodically run until the **PC** value is below the threshold limit value.

ALGORITHM 7: Battery Compensation Algorithm

```
1: procedure ADJUSTMENTBATTERYCOMPENSATION()  ▷ Adjust speed of the
   robot Input: robotMotorSpeedValue = UserInputValueForMotorSpeed
2:     //Use built in Microsoft Drive functions to update the motor speed value
3:     Drive.SetDrivePowerRequest request = new
   Drive.SetDrivePowerRequest()
4:
   request.LeftWheelPower = (double)OnMoveLeft * robotMotorSpeedValue;
5:     request.RightWheelPower =
   (double)OnMoveRight * robotMotorSpeedValue;
   end
```

6.6 Evaluation

Using the same task configuration used in Section 6.4.1.1, Table 6.6 shows the parameter values used for the evaluating the Battery Degradation case study. The equations used in Section 6.4.1.1 are applied to a range of 'Cycle Counts' that represents the life of the battery.

Table 6.6: Parameter values used in the evaluation of the battery performance using a DOD rate of 30 %.

Parameters	Values
RS Robot Speed (mm/s)	600 (0.6 M)
PR Power required (W)	8
DIS Distance to travel (M)	5,000
T Time (hr.)	2.31
WHU Watt Hours used	18.48

In Figure 6.8, the *percentage charge* needed from the battery to complete a robot task (see Table 6.6), decreases as the battery reaches its mid-cycle range (200-700). After the battery cycle has reached 800, the *percentage charge* **PC** needed increases as the battery starts to experience the effects of degradation. Eventually the *percentage charge* required exceeds the *Tolerance Limit* (1100-1150), and therefore the robot task cannot be completed using the current 'parameter' configuration in Table 6.6.

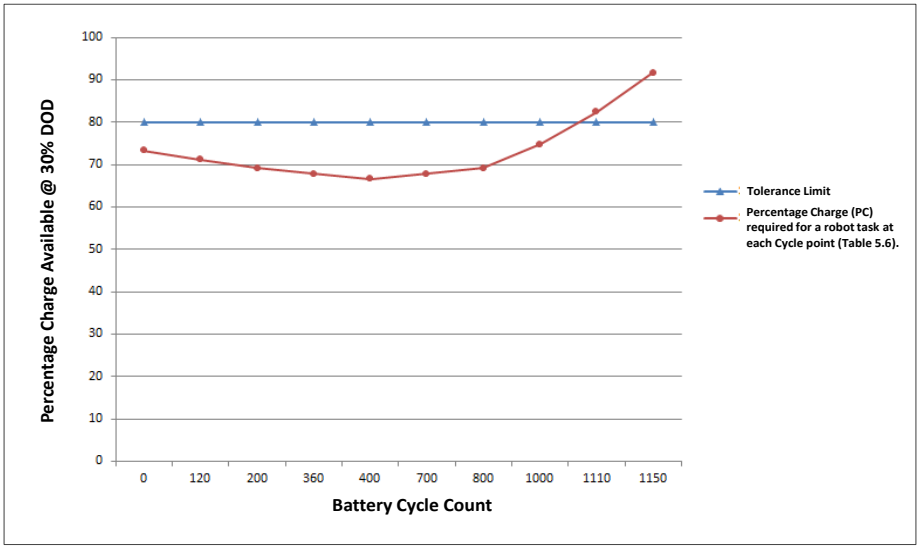


Figure 6.8: Line chart showing what Percentage Charge is available to a robot task at given cycle point within the battery lifetime.

In Figure 6.9, the *Battery Degradation Adjustment* policy is applied when the battery gradation fault has been identified (battery cycle 1000). In Figure 6.8, battery cycle 1100 and 1150 where above the *tolerance limit*. In Figure 6.9, the application of the *Adjustment Policy* brings battery cycle 1100 and 1150 below the *tolerance limit*.

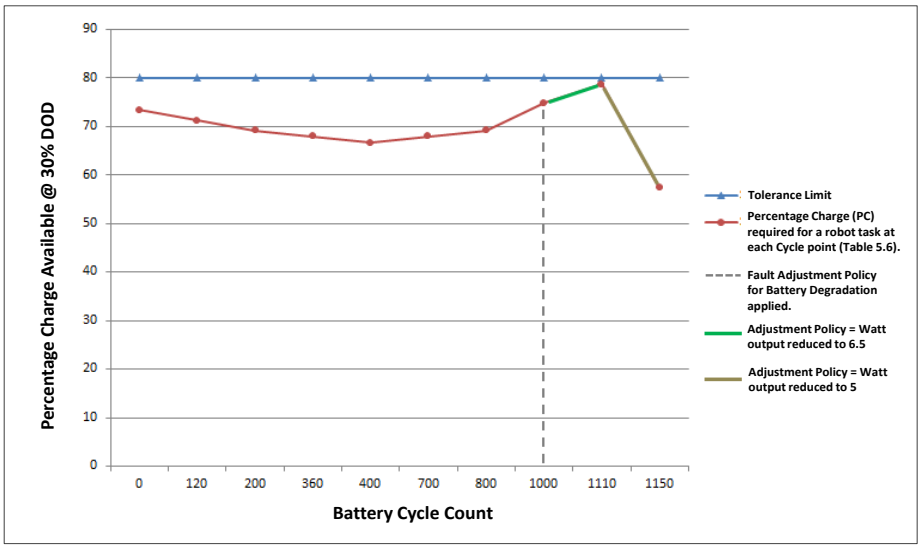


Figure 6.9: Line chart showing what Percentage Charge is available to a robot task at given cycle point within the battery lifetime.

Further development of the AIFH architectural model showed a *battery degradation fault* can be managed using both the robots System Manager and the Autonomic Manager. The three layers in the Autonomic Manager (Awareness, Analysis and Adjustment), interacted with the Reactive Control and Proactive Control feedback loops to handle the *battery degradation fault*. Using the Autonomic Manager Knowledge Base, policies where used to analyze the extent of the fault and to make adjustments to compensate for the fault. *Self-adjustment* allows the robot to complete tasks even when operating with reduced power availability. Ultimately the battery in a *degradation* state will need replaced. The goal in this Cast Study, was to allow the robot to function as long as possible until even with operating at low power.

6.7 Summary

Autonomic Management for Mobile Robot Battery Degradation case study set out to demonstrate how an Autonomic Management System can be implemented to handle power management faults in a mobile robot such as the Pioneer P3-DX. The System Development Life Cycle (SDLC) was implemented as a Research Model to describe a sequence of activities including requirements, design, implementation, testing and evaluation. The Research question posed for this case study stated - if the AIFH architecture (derived from the MAPE-K autonomic model and IMD model), can be applied to detecting power management faults on a mobile robot and if possible, compensate for those faults? Can the mobile robot adapt to its changed environment and continue to function even with a battery fault?

The AIFH architecture was expanded in order to handle power faults within a mobile robot. In this case study, tests where carried out on how battery degradation can affect the performance of a mobile robot over time. This type of fault scenario is predictable compared to the *Wheel Alignment Fault* and the *Sonar Sensor fault* case studies, in which a fault can occur at any time during a mission. Battery degradation is an unavoidable process and therefore the Autonomic Manager must adapt its policies to handle this type of disability. The AIFH 'Awareness' process in this case

study, is knowing when the battery degradation has begun. This relies on knowledge regarding the DOD that is being adapted (Fig. 6.2) and checking the *cycle count* of the lead-acid battery in the robot. The AIFH 'Analysis' process can cross referencing the cycle count with the percentage of charge available in the battery to establish if the *tolerance threshold* has been exceeded. The data from the 'Analysis' process is then made available to the AIFH 'Adjustment' process. The 'Adjustment' policies can then calculate what power reduction in certain components is required so therefore reducing battery consumption.

The research carried out in this case study and the previous case studies (Chapter 4 and Chapter 5), provides a valuable experience in gathering knowledge for the implementation of a Generic Autonomic Architecture for fault handling in a mobile robot (see Chapter 7).

Chapter 7

Generic Architecture for Fault Detection (AIFH)

7.1 Introduction

In this Chapter, the final design for the *autonomic intelligent fault handling* architecture (AIFH) is presented. In Chapter 3, Section 3.3, the initial concept of the AIFH architecture was introduced with the notion of a 3-layer architecture. The initial ideas for the architecture where based on a combination of the MAPE-K and IMD architectural models. The aim of the generic autonomic architectural design is to handle various types of component faults within a mobile robot. Through a series of 3 case studies, the architecture has been refined as experience and insight from the studies was gained. In Chapter 4, 5 and 6, case studies where employed to explore how different fault scenarios could be used to develop the AIFH architecture. The generic autonomic architecture or AIFH, is a triple layer model consisting of an *Awareness Layer*, *Analysis Layer* and *Adjustment Layer*. These three layers are controlled by an Autonomic Manager. The Autonomic Manager contains feedback loops that traverse through each of the layers. The System Manager controls the flow of data from the robot's sensors and effectors. The High-Level AIFH architecture is introduced in Section 7.3. The Low-Level detailed architecture is presented later in Section 7.6. The following Sections explain the roles and responsibilities of a number of the architectural components prior to seeing them within the overall

design architecture (Fig. 7.8). Evaluation of the generic architecture is presented in Section 7.8, through a further case study which explores the utility of the evolved AIFH architecture.

7.2 Overview - Generic Architecture (Fault Handling)

The design of software systems requires the ability to describe, create, and evaluate systems at an architectural level. Designing a reliable system involves integrating failure responses within the system. Failure responses can be categorized by stages. It is important that the architectural system design provides a response for each stage. Stages include *Fault Detection*, *Fault Diagnosis*, *Recovery and Repair* [113]. Faults occurring in systems often don't limited themselves to one component. Research carried out in [114], incorporates a generic fault tolerance architecture (GFTSA), to handle faults by using error recovery mechanisms in Distributed Systems. Their architectural fault tolerance proposal is designed so that it can be re-used at different levels within the Distributed System. This *re-usable* design is important within the AIFH architecture. If a fault occurs in the robot's sensors system, drive system or power management system, the AIFH architecture needs to be adaptive to handle these faults from multiple systems within the mobile robot. In designing a generic architecture, basic elements need to be incorporated such as re-usability, adaptive (reacting to requirement changes without the need for restructuring) and low complexity (complex systems require more maintenance and are more difficult to understand and use). Research in *Generic Architecture* in [115], proposes their GeRDI (Generic Research Data Infrastructure), that uses policies that can adapt to shifting requirements. By extrapolating requirements, they in-turn create domains that contain features that are *self-contained*. In designing the AIFH generic architecture, each fault scenario has its own set of requirements. If a fault is contained within a particular robot sensor, then the architecture needs to be flexible in identifying, analyzing and if possible, compensating for the fault. Equally, if the fault occurs in the 'power' system of the robot, a new set of requirements needs to

be processed using the same *generic* model. During a mobile robot's lifetime, it may receive various types of hardware upgrades i.e. new sensors etc. The AIFH *Knowledge Base* needs to flexible to incorporate new policies to handle these new components. Therefore, if a fault occurs in a relatively new component (sensors), the AIFH can adapt to its changed environment.

7.2.1 Comparative analysis of the architectural model used in each case study

The case study in Chapter 4 introduced a 3-layer concept to describe the AIFH architecture. The Awareness Layer would process the sensor data and identify fault patterns, the Analysis Layer would analyse the data to establish the extent of the fault, finally, the Adjustment Layer would then determine if a compensation strategy could be applied to the fault data, so that the robot could continue to function in some capacity. Chapter 4 also introduced a basic *feedback* loop, that transported the data between each layer and to the System Manager. The case study in Chapter 5 introduced the autonomic Knowledge Base, which contains different data types, that can be used by each of the three layers within the AIFH architecture. Compared to the previous case study in Chapter 4, the architecture in this case study can call on policies and historical data to handle faults. When a component failure occurs, the Autonomic Manager will select the appropriate recovery policy to compensate for the fault. The Knowledge Base also contains *dynamic parameters* that can be adjusted if environmental changes occur. The final case study (Chapter 6), explores the use of *reactive* a proactive feedback loops. This allows the Autonomic Manager the ability to report possible pending faults by comparing historical and real-time data.

7.3 High-Level AIFH Architecture

When building an Autonomic Architecture, certain characteristics are required to define the autonomic system. The autonomic system is required to hold *self-knowledge* and therefore acquire knowledge of the surrounding environment. The autonomic system is required to detect and recover from component failure and

therefore, maintain a level of independence. In Figure 7.1, the basic AIFH architecture consists of an *autonomic manager* with connections to hardware systems via the *system manager*. Within the *autonomic manager*, the *self-healing* module is responsible for initiating the feedback loop that transports data through each of the AIFH layers. The 3 layers (awareness, analysis and adjustment), all have access to the *knowledge base*.

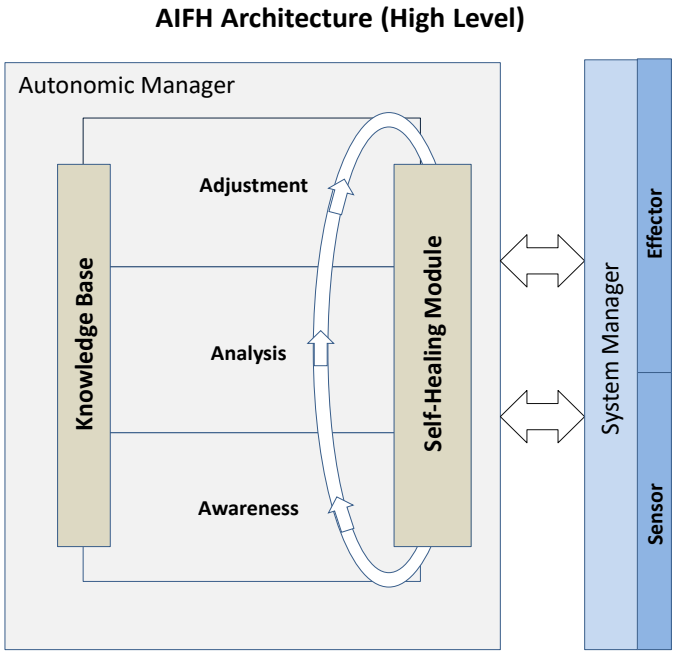


Figure 7.1: AIFH architecture (High-Level view).

7.4 AIFH architectural components

7.4.1 System Manager

The System Manager is responsible for controlling the sensors and effectors of the mobile robot. The System Manager feeds data from the sensors to the Autonomic Manager. The Autonomic Manager can then process the sensor data and also update the Knowledge Base module - see Figure 7.2.

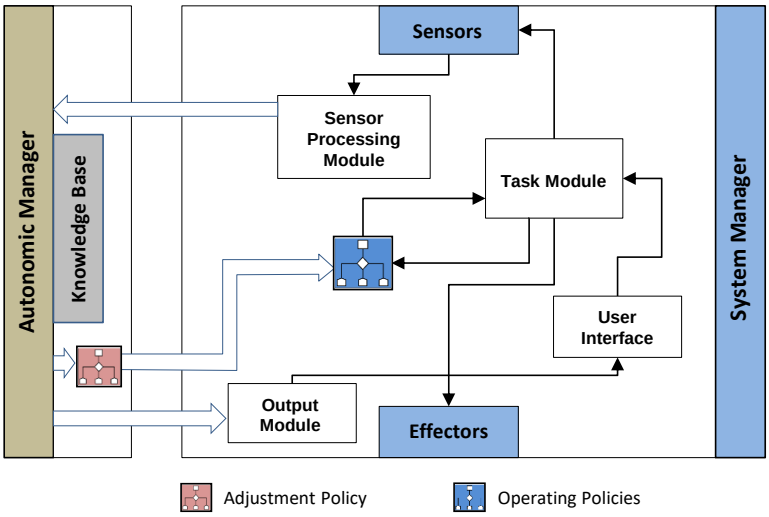


Figure 7.2: AIFH architecture - System Manager modules.

The *Task Module* within the System Manager, is used to process 'task' data inputted from the User Interface and to execute commands taken from the *Operating Policies*. The *Task Module* initiates the robot sensors and sends command operations to the effectors. Data from the robot's sensors is processed using the *Sensor Processing Module*. The *Sensor Processing Module* is responsible for sending data to the Autonomic Manager; this data contains sensor readings accumulated during the task operations. The Autonomic Manager, if required, can update the System Manager *operating policies* via the AM's *adjustment policies*. These *adjustments* would be in response to a *fault* status, within the mobile robot. If a *fault* status is apparent, then the Autonomic Manager needs to make adjustments to the current *task* by changing the behavior of the *operating policies* within the System Manager. The System Manager also contains an *Output Module* which is used to relay data supplied by the Autonomic Manager's (Knowledge Base module), to Users or Mission control, regarding fault diagnosis, symptoms of possible impending faults and fault recovery information.

7.4.2 Autonomic Manager

The Autonomic Manager implements the feedback loops and makes use of domain-specific knowledge to process task data supplied by the System Manager [49]. The

Autonomic Manager makes use of monitored data from sensors and combines this with stored knowledge to plan and implement tasks. The Autonomic Manager, if required, is responsible for initiating behavioral changes regarding the execution of tasks. Behavioral changes are usually the result of an environmental change i.e. a *fault* status is declared. The Autonomic Manager would then relate these *changes* or *adjustments*, to the System Manager's *operating policies*. In terms of fault management, the Autonomic Manager benefits from using previous experiences or historical data, to identify trends in component behaviours. When a fault occurs within a component, it may not necessarily be represented as a straightforward *fail* state. Some faults within components may not be discernible to begin with, but over time, the failure gradually becomes more evident. The Autonomic Manager must not only report (via the Output Module - see Figure 7.2), if a component suffers catastrophic failure but also report on the behavior of a component that is under performing or showing a gradual downwards trend.

7.4.2.1 Self-Healing Module

Within the Autonomic Manager, the *Self-Healing* module is responsible for monitoring, detecting and diagnosing system malfunctions. It achieves this by initiating the feedback controls loops. In the AIFH architecture two control feedback loops are employed (Reactive and Proactive). Research developed in [116], shows that coordinated parallel control loops can be used to carry out separate operations as long as each control loop does not violate the objective of another controller. The Reactive Control loop passes data between each of the 3 Layers (Awareness, Analysis and Adjustment). The Reactive Control loop is part of the Autonomic Manager that immediately identifies a component failure. There is no requirement for adapting experience or historical data in this instance, as identifying and analyzing the fault needs to be carried out quickly. The Proactive Control loop operates within the Awareness and Analysis Layer. The Proactive Control loop will use past experience and historical data to investigate component anomalies. The Proactive Control loop is part of the Autonomic Manager that reports any particular trends in a component's performance. The Autonomic Manager also contains the *Knowledge Base*

module. The Knowledge Base module is available to all 3 Layers within the Autonomic Manager.

7.4.2.2 Knowledge Base Module

The information stored in the Knowledge Base can be used to extend the knowledge capabilities of an Autonomic Manager. The *Knowledge Base Module* provides each layer in the AIFH architecture with tolerance values, polices, historical data, dynamic parameters, real-time data and symptoms.

- **Tolerance Values** - task performance is measured against expected result markers. Tolerance values are used so that tasks are completed within an established level of accuracy. If a Tolerance value is exceeded, then the task has failed.

- **Policies** - an Autonomic System requires a method for defining polices that allows the Autonomic Manager to make informed decisions [47]. Implementing policies in a standard way, means the whole autonomic system can be managed using a common set of policies. In the AIFH architecture, *awareness*, *analysis* and *adjustment* policies are stored within the Autonomic Manager. The System Manager stores the *operating* policies that used for *command* routines for the sensors and effectors.

- **Historical Data** - as each robot task is performed, the sensor data is recorded into the Knowledge Base. The Autonomic Manager uses historical data to identify any significant changes in the performance of sensors over time. Historical data is also useful for identifying trends within task data.

- **Dynamic Parameters** - are sets of parameters that are predefined but can be updated when required. For example, if the battery *cycle count* (in the mobile robot), reaches a certain level, then this is compared to the dynamic parameter value: if they are equal, then an alert message is sent to the AM.

- **Real-Time Data** - real-time data is information currently being reported by sensors during a task. The Autonomic Manager may need to react quickly, if

a sensor is reporting irregular data, i.e., a sensor is damaged in a collision.

- **Symptoms** - this is part of the Knowledge Base that records any possible impending issues with components. Symptoms can be identified within task performance data. Even though sensor data is within Tolerance limits, reports can show a downward trend that might indicate a possible fault in the future.

The *Knowledge Base Module* is dynamic and is constantly updated with sensor data supplied by the executing task. Figure 7.3 shows how each attribute within the *Knowledge Base Module* is used by each Layer within the AIFH architecture.

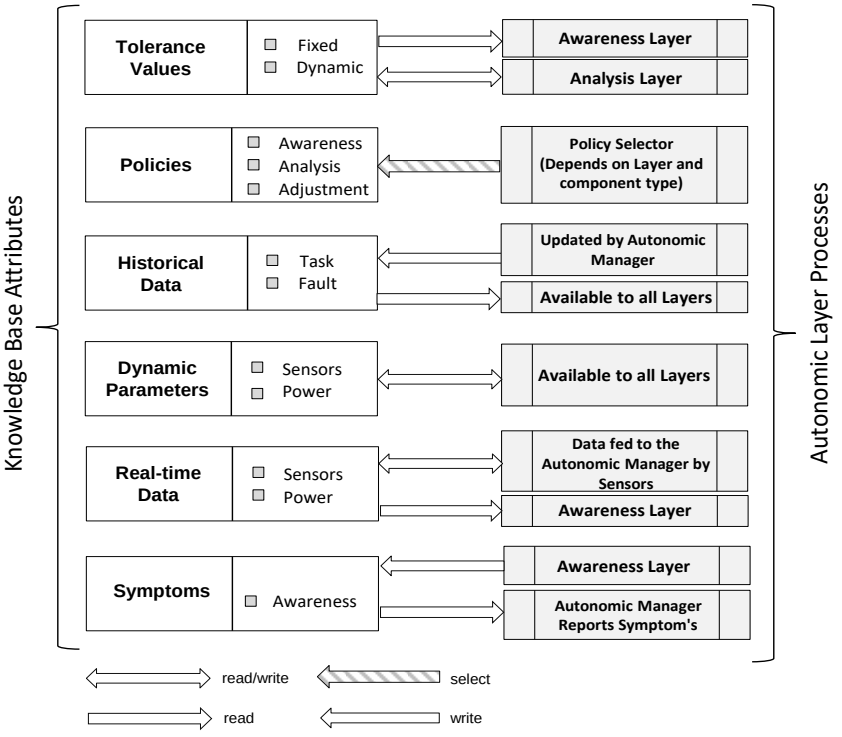


Figure 7.3: Shows how the attributes within the Knowledge Base are used by each Layer within the AIFH Architecture.

Tolerance values within the *Knowledge Base Module* can be read by both the *Awareness Layer* and the *Analysis Layer*. The 'tolerance' value *process* works on a number of levels. When a task is being executed by the robot, data from multiple sensors are being processed by the Autonomic Manager (*Awareness Layer*). Depending on what type of task is being performed and what sensor is being used, will dictate the type of 'tolerance' value. In Figure 7.4, tolerance type is described as

(TT). The sensor/historical data is checked using the Check Tolerance Value **(CTV)** algorithm. Depending on the value within data, the CTV algorithm will

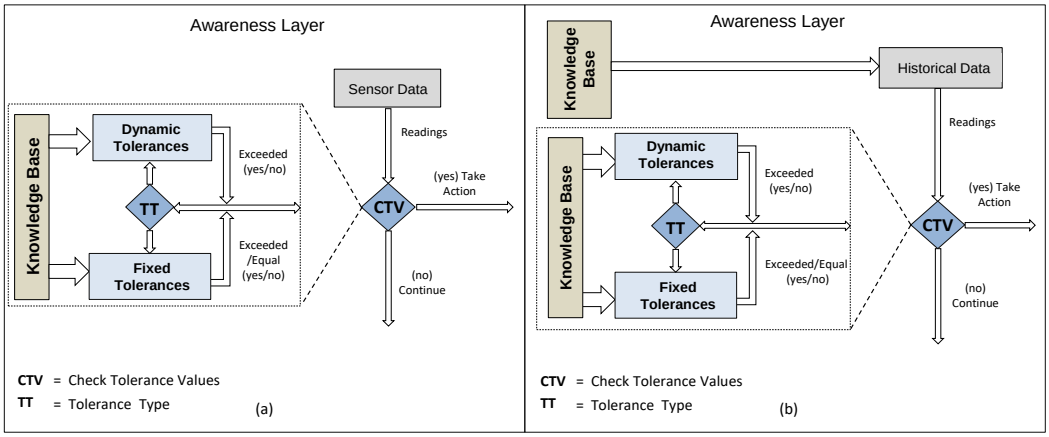


Figure 7.4: (a) - Shows tolerance values compared to real-time sensor data. (b) - tolerance values compared to historical data.

The tolerance type (**TT**), can either be 'dynamic' of 'fixed'. The selected *tolerance* value is then used against the current sensor reading. If the sensor reading is above the expected *tolerance* value then action is required, and the sensor data is 'tagged' for further investigation.

Sonar Object Detection - P3-DX robot		Results
Sonar Sensor #	Readings (object reflection) mm	Tolerance 'fixed' disabled sensor = 5000
0	878	FALSE
1	1089	FALSE
2	5000	TRUE
3	5000	TRUE
4	5000	TRUE
5	5000	TRUE
6	2567	FALSE
7	1297	FALSE

Figure 7.5: Example of 'fixed' tolerance value - used to identify disabled sonar sensors

An example of a 'fixed' tolerance value is shown in Figure 7.5. The sonar sensors on a P3-DX Pioneer robot defaults to a reading of '5000' when in a *disabled* state. The 'fixed' tolerance value is also set to '5000'. If sensor reading equals the tolerance value, the sensor would be declared as faulty.

Tolerance values can also be used on *historical* sensor data (see Figure 7.4 (b)). If a robot is performing a repetitive task, the sensor data is recorded into the

Knowledge Base. As part of the *self-healing* process, the historical data is periodically monitored. The historical data is then compared with the accepted 'tolerance' values. If the tolerance value has been exceeded, then that section of historical data is 'tagged' and sent for further investigation to establish the extent of the fault.

An example of a tolerance value being used with historical data, is shown in Figure 7.6. The mobile robot is expected to arrive at its destination point within a tolerance value (25mm). If the tolerance value is exceeded, then this is flagged as a possible fault. This tolerance value can be altered if required and is therefore described as being 'dynamic'.

Distance (mm) robot is from expected destination point										Results	
Test1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10	Average distance from expected Destination point mm	Tolerance within +- 25mm
2	9	-9	4	5	-7	-2	7	-6	-5	-0.2	TRUE
-8	-13	-13	-11	-14	-4	-13	2	-6	-6	-8.6	TRUE
3	-21	-12	-10	4	-10	9	-11	-4	-21	-7.3	TRUE
-25	-30	-33	-44	-45	-20	-33	-39	-22	-59	-35	FALSE

Figure 7.6: Example of a 'dynamic' tolerance value for laser sensor *distance* readings

The *Awareness Layer* can also update tolerances if a particular tolerance value is too sensitive. For example, there is a tolerance value used to check if two sonar sensors are showing the correct distance reading when detecting an object - as the sonar array on the robot is octadecagon, this tolerance value may need to change depending on the angle of the robot to the object.

The 'Policy Selector' process (see Figure 7.7), is used by all 3 layers in the AIFH architecture. Some policies are used to check if sensors are operating within tolerance limits. Other polices involve analyzing data to establish the extent of a fault. Polices are also available that can adjust the behavior of the robot to compensate for a fault. In Autonomic Computing, policies can be described as *action*, *utility* and *goal* [54], - explained in detail in Chapter 2, Section 2.2.2.2. In the AIFH architecture, the policy selection process depends on the type of fault that has been identified. If we take an example for a sonar sensor fault (as investigated in Chapter 5), various policies are required to identify and handle the fault process - see Figure 7.7.

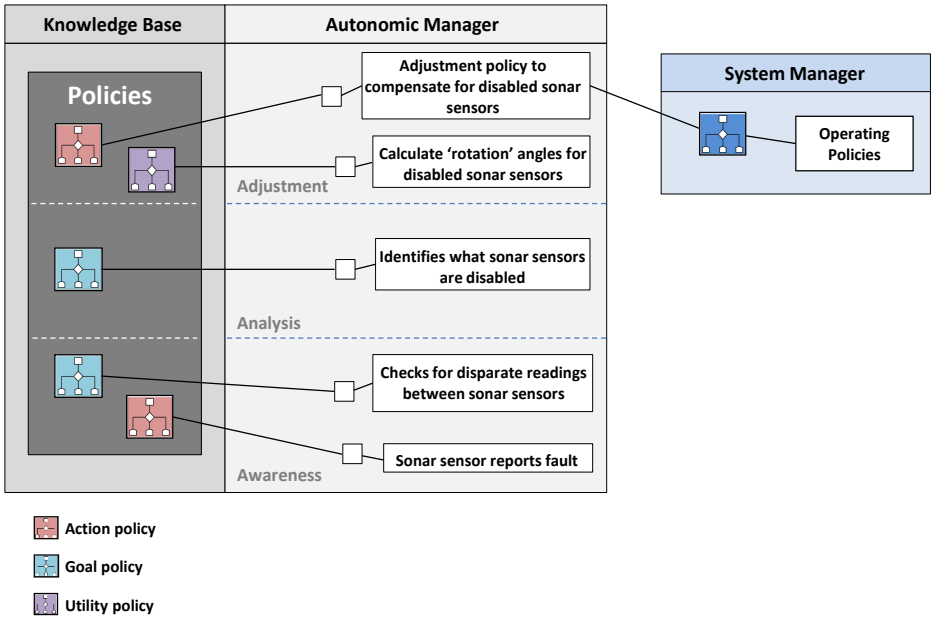


Figure 7.7: Policy Selector - *Knowledge Base* policies for Sonar Sensor Fault.

In Figure 7.7, the Autonomic Manager (Awareness Layer), contains an *Action* policy and *Goal* policy. The *Action* policy is required as the Autonomic Manager needs to react to the current state of the system. The sonar sensor has reported a fault *state*, therefore, action needs to be taken quickly. The *Goal* policy is more measured. In this case, the exact *state* is unknown, so this policy executes a number of checks between adjacent sonar sensors to establish if a particular sensor requires more investigation. The Autonomic Manager (Analysis Layer), contains a *Goal* policy to establish what actual sonar sensors are faulty as identified by the *A*wareness layer. The Autonomic Manager (Adjustment Layer), uses a *Utility* policy to calculate the angle of 'rotation' required by the robot to compensate for each disabled sonar sensor. Finally, a further *Action* policy (see Figure 7.7), is used to make the adjustments to the robot when executing its tasks and therefore compensate for any disabled sonar sensors. This action will update the *Operating Policy* within the System Manager.

The *Knowledge Base* Historical Data is constantly updated by the Autonomic Manager with sensor data supplied by the System Manager. When a robot task is completed, then the results of the task are recorded for reference purposes. For

example, a task that was executed by the robot required 15% of the robot's battery charge. If a similar task is required, then historical data is useful in deciding if there is enough power currently in the batteries to complete the task. Historical data is also important in order to track behavioral changes within the robot's components. Some components may degrade over time and this can have an impact on a robot while its executing various tasks. Dynamic Parameters are used to aid analysis when checking values against *Real-time* data. For example, the battery *cycle count* is a dynamic parameter that is updated every time the robot's battery is charged. The *Knowledge Base* Symptoms, records unusual readings from selected components. *Symptoms* are only recorded if they are within *tolerance limits* but are showing a *behavioral pattern* that may suggest a future impending fault. An example of 'Symptom' behaviour - when a robot is asked to conduct a series of tasks leading it to a particular destination, as it reaches the end of the tasks, the distance it is from the destination marker is progressively increasing. Even though the robot is operating within *tolerance* values, this behavior could indicate a progressive 'drive' system failure or minor damage to one of its wheel assembly's, that, when the tolerance value is eventually reached, will cause the analysis of the wheel alignment to be conducted with a view to changing the 'operating policy' for such tasks.

7.4.2.3 Keeping the Autonomic Manager focused

One of the aims in designing an autonomic architecture is to make sure the *autonomic procedures* themselves do not overwhelm the system that is being monitored. If the Autonomic Manager takes on vast amounts of sensor data from the robot, then it will consume the same volume of CPU time and memory capacity as the System itself [49]. Therefore, the data drawn in by the Autonomic Manager should be optimized in a way that the AM can receive the right enough data, so it can make an informed decision, if it finds the data is showing anomalous behavior.

7.5 Building the AIFH Architecture

In Chapter 2, attributes from both the MAPE-K and IMD architecture where used to build the AIFH basic architectural design. In Chapters 4, 5 and 6, the case stud-

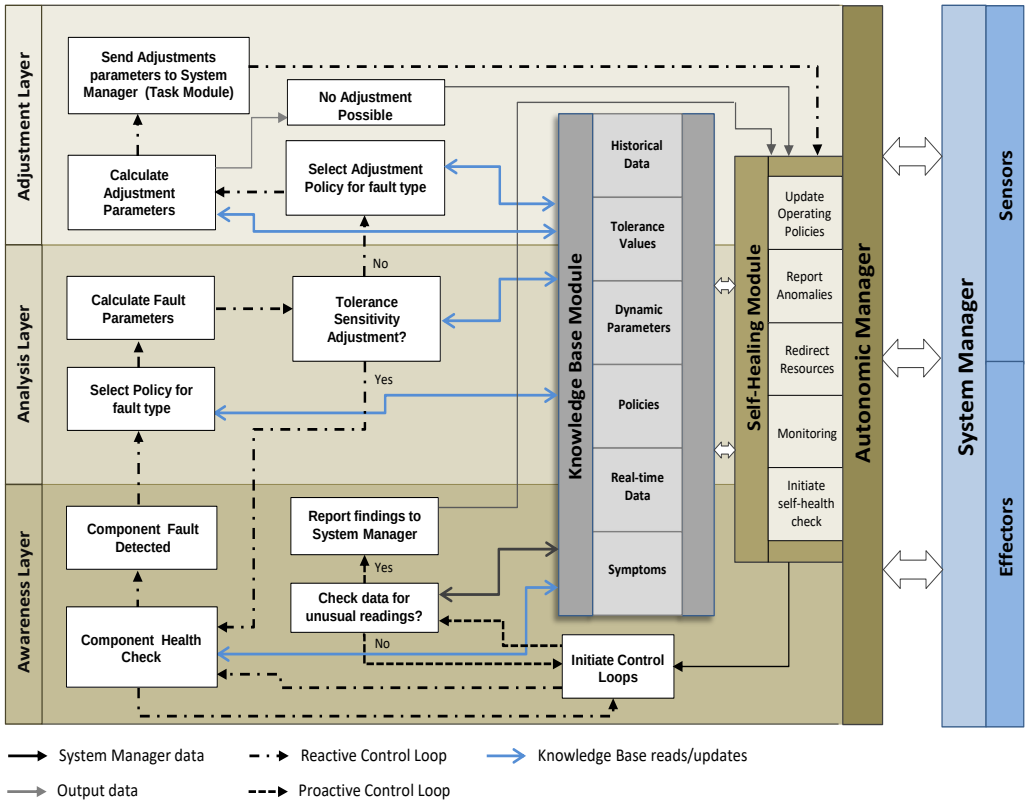


Figure 7.8: Low-Level AIFH Generic Autonomic Architecture.

ies provided methods for detecting and analyzing faults. There was also methods to adjust for those faults. These experiments provided a foundation to develop the AIFH architecture in greater detail. In Fig. 7.8, the Autonomic Manager is presented with the Knowledge Base module and connection to the System Manager, as a fully formed Generic Autonomic Architecture. The Autonomic Manager (*self-healing* module), controls the timing of the *Health Check* monitoring and initiates the *feed-back* loops that traverse each layer (*Awareness, Analysis and Adjustment*). The Knowledge Base is shared by each of the AIFH autonomic layers and includes policies that can be used to detect and adjust for component faults. In this Section, each of 3 Layers in the AIFH generic architecture are explored in detail, showing how they interact with each other and how they interact with the Knowledge Base module.

7.5.1 Low-Level AIFH Architecture

In Section 7.3 the High-Level AIFH architecture was presented. In this Section, the AIFH architecture is explored in greater detail at its lowest level. The Autonomic Manager is largely independent (*self-managing*), however, it relies on communication with the System Manager for incoming sensor data. The Autonomic Manager will also communicate with the System Manager if changes are required to handle component faults - see Figure 7.8 - (Update Operating Policies). One of the properties of an autonomic system is *self-healing*. *Self-Healing* is presented in the AIFH architecture as a module that controls the autonomic feedback loops and controls the data output that the System Manager requires if changes have to be made to alter system commands for fault compensation. In autonomic terms, *self-healing* is the ability to find, diagnose and react to system malfunction. To discover system malfunctions or possible future faults, the autonomic system must have knowledge about its own behaviour and knowledge of the local environment [117]. The *self-healing* module monitors for any apparent changes to the local environment. It monitors systems by initiating *self-health* checks. The data from the robot sensors is exposed to the Awareness Layer. However, as discussed in Section 7.4.2.3, this data is selected at *intervals* to ensure it does not overwhelm the System. This Layer will either report back to the *self-healing* module that no faults have been discovered or it will identify component failures and report this data to the Analysis Layer for further processing. The AIFH architecture has been designed in such a way, that each layer within the Autonomic Manager has access to the Knowledge Base and therefore, using appropriate policies, can make decisions in terms of how to manage system faults.

7.5.2 Awareness Layer

As the mobile robot executes its tasks, the Autonomic Manager will periodically check on the health and functionality of the hardware components. We define *Awareness* as the ability to detect that the data being processed and monitored may be indicating a possible fault. The Reactive Control loop initiates a health check on all components that are to be used for the current robot task - see Fig. 7.8. This

can involve *detection sensors, cameras, motor differential drive and power supply*. Tolerance values held in the *Knowledge Base Module* are used to indicate if there is a possible issue with a component. If tolerance values are exceeded, then this can indicate a possible fault. If, for example, a sensor unit is reporting a *disabled* state, then the Reactive Control loop will relay this information to the *Analysis* layer for further processing. The *Awareness* layer can also process historical data and compare this with real-time data reported by the current task. The Proactive Control loop checks this data for patterns that might indicate a possible future fault. For example, if the robot completes a task that involves traveling from destination A to destination B, when doing a *self-check*, it finds that it is not exactly at point B but is still within tolerance limits. However, if this trend continues in further tasks, then it might be an indication that a wheel fault is about to occur. The Proactive Control loop is responsible for reporting unusual data readings to the *Self-Healing* module. The *Self-Healing* module will send these reports via the Output Module (in the System Manager), to the User Interface or Mission Control. These reports are vital and could prevent future tasks being compromised. Figure 7.9 shows an UML Sequence representation of the modules and process routes within the Awareness Layer.

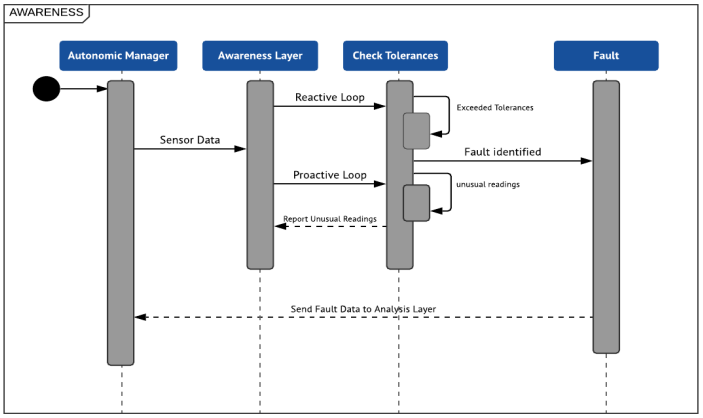


Figure 7.9: UML diagram showing the relationships within the AIFH 'Awareness' Layer.

7.5.3 Analysis Layer

Through analysis, we can establish the extent of a fault indicated in the *Awareness* Layer. Depending on the type of component identified, the relevant analysis policy is selected from the *Knowledge Base Module* - see Fig. 7.8. The analysis policy (see Policy Selector - Section 7.4.2.2), will then determine the extent of the fault. Calculations are performed using the analysis policy, which are then passed to the *Adjustment Layer*. For example, if a sonar fault has been identified in the *Awareness Layer*, then an analysis policy can determine how many of the sonar sensors on the array are disabled. Specialized policies can determine if the sonar sensor is reporting the correct *distance* data by comparing results with adjacent sonar sensors. Other examples include *wheel alignment policies*. If the *Awareness Layer* determines there is an alignment fault, then a policy can be used to determine how much the robot's alignment is from the expected *true* alignment. The value returned would be labelled as the *offset* value. The *offset* value can then be passed to the *Adjustment Layer*.

Another property of the *Analysis Layer* is the ability to determine if current *tolerance* values are too sensitive. If a tolerance value is set too 'high', then this can result in the *Awareness Layer* reporting a fault during the next Autonomic *feedback* loop process. The *Analysis Layer* can make the necessary adjustment to the tolerance values if required. For example, if a *wheel alignment* tolerance value is set in the *Knowledge Base* as 10 meters. Then this might need to change if the terrain the robot is operating in, prevents the robot arriving at a destination with any significant accuracy. The *wheel alignment* tolerance value could then be adjusted to 20 meters.

If *tolerance* adjustment is required, the Reactive Control loop will re-direct back to the *Awareness Layer* for re-evaluation. Once the fault calculations are made in the *Analysis Layer*, then the fault parameter data is passed to the *Adjustment Layer*. Figure 7.10 shows an UML Sequence representation of the modules and process routes within the Analysis Layer.

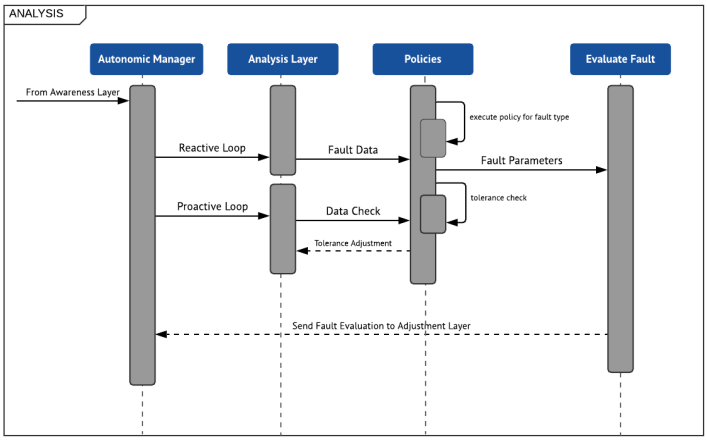


Figure 7.10: UML sequence diagram showing the relationships within the AIFH Analysis Layer.

7.5.4 Adjustment Layer

Using the fault parameter data supplied by the *Analysis Layer*, the *Adjustment Layer* will select the appropriate *adjustment policy* from the *knowledge Base* module. Calculations are then performed so that a compensation strategy can be employed to handle the component fault - see Fig. 7.8. Adjustment calculations may determine if there can be no resolution to the current fault. For example, if all sonar sensors are reported as being *disabled*, then the sonar sensor array cannot be used to detect objects along the robot's path. In this case, a message is sent to the System Manager (output module) to report that no compensation can be made to the reported fault. However, if an adjustment calculation can be made, i.e., if there is at least one sonar sensor still operable, then an *adjustment* policy can be deployed.

When the *adjustment* policy is deployed, the Reactive Control loop will send the *compensation* parameters via the Autonomic Manager to the System Manager (Operating Policy). The *Operating Policy* will then update the System Manager (Task Module), which in turn, will direct the effectors/sensors to operate using the new 'adjusted' settings. To test the *compensation* strategy is successful, the *Self-healing module* will then re-initiate the Control Loops in the Autonomic Manager (Awareness Layer). The *Component Health Check* module in the *Awareness Layer* will check the component against current tolerance limits. If the *adjustment* pol-

icy is successful, the Reactive Control loop will send data back to the *Self-healing* module in the Autonomic Manager to report that no faults are pending. Figure 7.11 shows an UML Sequence representation of the modules and process routes within the Adjustment Layer.

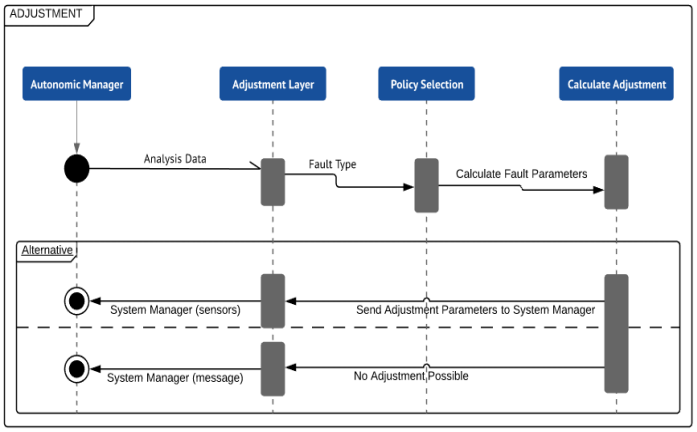


Figure 7.11: UML sequence diagram showing the relationships within the AIFH Adjustment Layer

7.6 Applying the Generic AIFH Architecture (Stereo Vision Camera Fault)

7.6.1 Introduction

The first 3 case studies where employed in order to expand our knowledge and explore the issues involved. In doing so, the AIFH architecture evolved to the extent that has been documented in this Chapter. In order to attempt some form of validation on the AIFH architecture, another case study was made. However, on this occasion, the intention was primarily to evaluate the completeness and feasibility of the AIFH architecture, rather than to evolve and refine it (as had been the objective of the first 3 case studies).

To evaluate the design of the AIFH generic architecture (see Fig. 7.8), we have applied it to a further case study centred on hardware faulting within a Stereo Vision Camera sensor. The overall objective was to demonstrate the utility of the generic architecture to a new fault scenario. The aim was to use all the layers within

the Autonomic Manager (Awareness, Analysis and Adjustment), to establish if a fault was occurring and if possible, make policy changes and *self-adapt* the System to compensate for the fault and thereby provide a first stage of feasibility check on our architecture.

7.6.2 Stereo Vision Camera - properties

The Stereo Camera can be used to identify obstacles and evaluate their distance from the robot. Fig. 7.12 shows a PCI nDepth Stereo Vision camera. This stereo vision camera and processing PCB board provides depth measurements by using a pair of sensors and a technology called computational stereo vision.

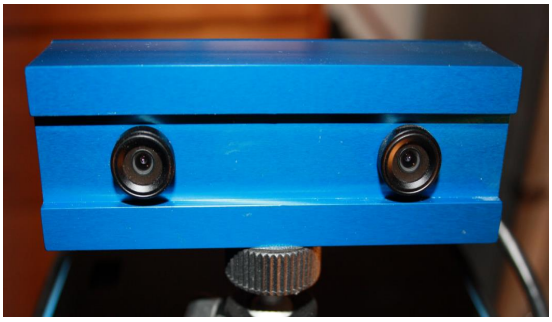


Figure 7.12: The PCI nDepth Stereo Vision Camera

The stereo camera provides real-time 3D depth data for mobile robot navigation. Evaluating distances is achieved as follows:

7.6.3 Triangulation

Figure 7.13 (a) shows a PCI nDepth Stereo Vision camera mounted on top of a Pioneer P3-DX. The basis of the technology is that a single physical point in 3-Dimensional space projects unique images when observed by two separated cameras. Figure 7.13 (b), shows a position **P** in 3D space and its projection to a unique location **SL** in the *left* image and **SR** in the *right* image. If it is possible to locate these corresponding points in the camera sensor images, the location of point **P** can then be established using *Triangulation*. The value **BL** represents the Baseline distance between the two sensors (in this case 6cm) and **f** represents the focal length of the sensors.

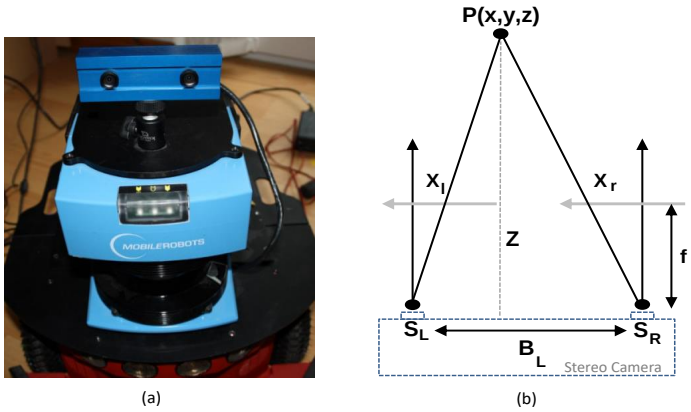


Figure 7.13: (a) The PCI nDepth Stereo Camera mounted on a P3-DX mobile robot. (b) Shows the Triangulation method for finding point P.

7.6.4 Disparity

Disparity is achieved by observing an object from slightly different perspectives. The position of an object in one image will be shifted in the other image by a value that is inversely proportional to the object and the stereo camera baseline [118]. As the distance from the cameras increases, the disparity decreases, this is useful for depth perception in stereo images. Points that appear in 2D stereo images can be mapped as coordinates in 3D space. This method was adapted by NASA's Mars Exploration Rover for scanning the surrounding terrain for obstacles. The rover uses its stereoscopic navigation cameras to capture a pair of images. Disparity calculations are performed in order to detect objects within the rover's path [119].

7.6.5 Awareness (finding a potential fault)

To calculate the distance between the camera and a known object, the Triangulation Stereo Vision method can be implemented [120]. The Stereo image pair consists of two images (left and right) and both images are combined to establish the disparity values and from that a **Z** distance value can be calculated from a selected object. However, this calculation can be affected by faults with the stereo camera sensor. Figure 7.14 shows the possible faults that can occur for a sensor in a stereo vision camera setup. If the stereo camera where to lose both sensors, then this could be picked up as either, no data being received from the camera or the robot 'bumper'

sensor being triggered by hitting an unseen obstacle, for example.

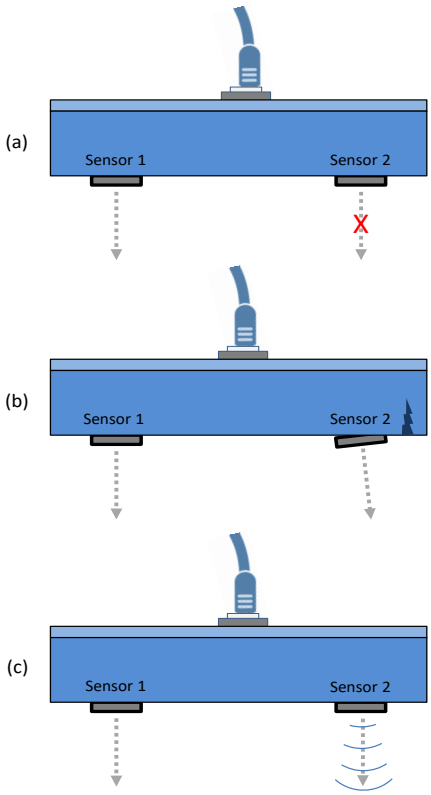


Figure 7.14: Stereo Vision Camera Faults. (a) Sensor shutdown, (b) Impact (pitch/yaw) and (c) De-focus Blur

In the AIFH architecture (Figure 7.8)), *Awareness* initiates monitoring and knowledge-based evaluation in order to establish if there is a potential fault with a hardware component. In Figure 7.14 (a), sensor 2 on the stereo camera is electronically disabled and cannot produce images for depth calculations. In this instance, we simply pass the *status* of sensor 2 to the Analysis process, where it will be labelled as disabled. In Figure 7.14 (b), the stereo camera has suffered an impact in the field; this has resulted in sensor 2 losing pitch/yaw relative to the stereo camera plane. Applying equations calculated by research developed in [121], we can establish that there is a depth error occurring in sensor 2. This is characterized by the size of the *yaw* angle between the two cameras. The greater the *yaw* angle the greater the depth error. This error data is then sent to the Analysis process. Figure 7.14 (c), shows how defocus blur can potentially influence the quality of the

disparity estimate. Research conducted in [122], explains how defocus can lead to objects appearing blurry in the image. Therefore, we can apply equations calculated in [122], to establish if a sensor in the Stereo Camera is exhibiting defocus error characteristics. If this is the case, we can send the error data to Analysis for processing. In the real world, fault scenarios shown in Figure 7.14 (b) and (c) will not indicate what sensor has failed. To establish what sensor has failed, will require in-depth analysis.

7.6.6 Analysing (establishing what sensor is faulty)

From the *Awareness* process carried out in Section 7.1.5.5, we need to establish the extent of the fault that has been discovered. The AIFH Architecture (see Figure 7.8), shows how component analysis is carried out using information gathered from the *Awareness* process. The Analysis process has specialized algorithms which can be used to identify the extent of the component fault.

For the fault indicated in Figure 7.14 (a), there is only a requirement to set the state of the faulty sensor to *disabled* and then send this information to the *Adjustment* process. For the faults discovered in Figure 7.14 (b) and (c), then we need to carry out a calibration process to establish what sensor on the stereo vision camera is faulty. To carry out the calibration, we need to establish the actual distance between the stereo camera and the object. As faults can happen in the field, we need to use the mobile robot to establish this *distance* value. We can achieve this by using the *Bumper* sensor mounted on the front of the mobile robot. To establish the distance value, we drive the robot towards the object. We record the distance covered by the robot as it moves (using wheel encoder values). When the object meets with the *Bumper* sensor, the robot will automatically stop. Figure 7.15 shows how the Pioneer P3-DX robot can be used to measure the distance between the stereo camera and the object.

1. **ed** - wheel encoder distance (recorded as the robot drives towards the object).

2. **bb** - bumper baseline (the distance between the differential drive base line

and the bumper baseline).

3. **sb** - stereo camera baseline (the distance between the differential drive base line and the stereo camera baseline).

4. **d** - distance to object from stereo camera sensor baseline.

Using equation (7.1) we can calculate distance **d**.

$$d = (ed + bb) - sb \tag{7.1}$$

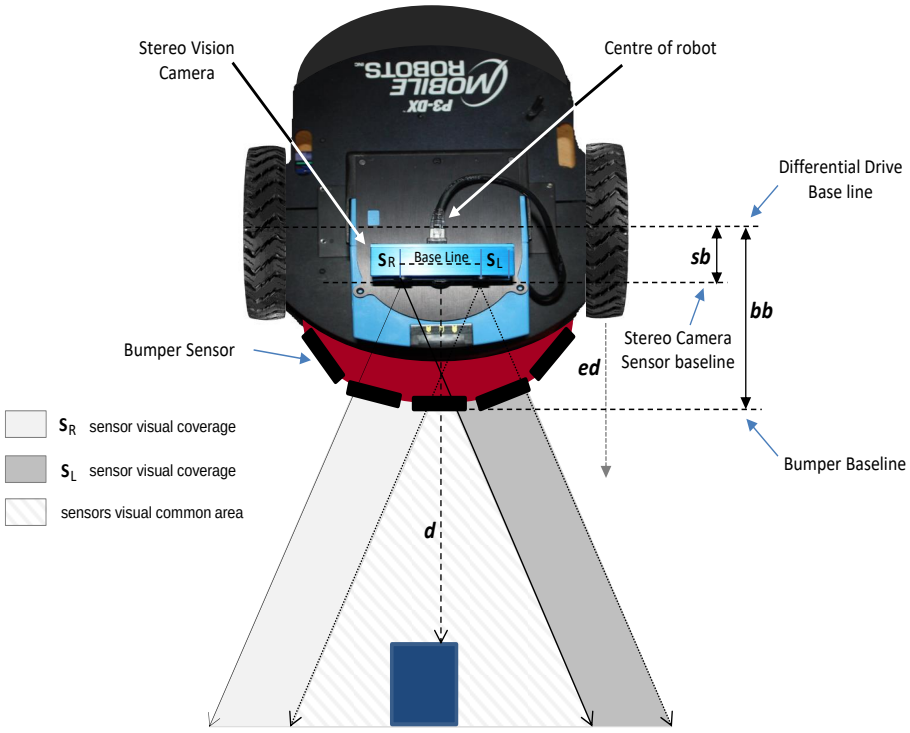


Figure 7.15: The Pioneer P3-DX Bumper can be used to calculate the distance between the stereo camera and the object

Now we have established the distance between the stereo camera and the object, we need to determine what sensor on the camera is faulty. There are a several scenarios (see Table 7.1).

From the fault scenario establish in Figure 7.14 (b) and (c), we can assume that both camera sensors are providing data. Therefore, for this experiment we can

Table 7.1: Fault Scenarios

No.	Camera Sensor (left)	Camera Sensor (right)	Comments
1	disabled	disabled	Both camera sensors are reporting no data
2	disabled	data (good)	Left camera is disabled, Right camera is providing reliable data
3	data (good)	disabled	Left camera is providing reliable data, Right camera is disabled
4	disabled	data (bad)	Left camera is disabled, Right camera data is unreliable
5	data (bad)	disabled	Left camera data is un-reliable, Right camera is disabled
6	data (bad)	data (bad)	Both cameras are providing data that is unreliable
7	data (good)	data (bad)	Left camera is providing reliable data, Right camera data is unreliable
8	data (bad)	data (good)	Left camera data is unreliable, Right camera is providing reliable data right

concentrate on testing scenarios 6-8, in Table 7.1 only.

To test scenarios (6-8) in Table 7.1, we must evaluate each camera's sensor individually. The analysing procedure carried out by the Autonomic Manager in Figure 7.16, involves using a specialized policy to test each individual camera sensor. The procedure involves taking a picture with camera sensor SL (see Figure 7.15), then storing this data. We then move the robot so that sensor SL is in the exact position where sensor SR should be. We then take another picture (current image). We then apply the Triangulation Stereo Vision method from [120] using the stored image and the current image to establish the *distance to object* value.

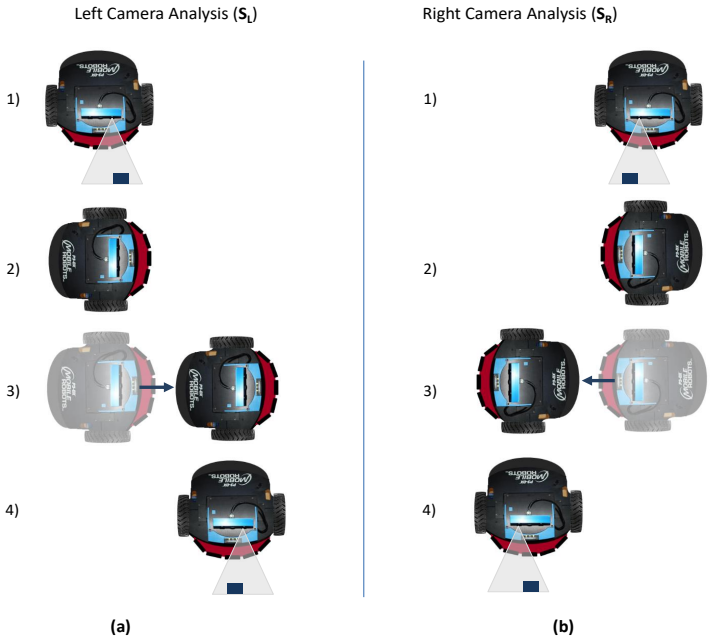


Figure 7.16: Shows how each camera sensor can be tested by evaluating two images taken by the same camera sensor from its original position and from the position of the opposing camera

Steps required for sensor evaluation - see Figure 7.16

1. Take a picture of the object with one camera only (a) or (b) - see Figure 7.16, then Store image data.

2. Rotate the robot 90°or -90°depending on what camera sensor is being evaluated (Figure 7.16 (a) or (b)).

3. Move the robot forward a distance equivalent to the Stereo Camera *Base Line* value between sensor SL and sensor SR - see Figure 7.15.

4. Rotate the robot 90°or -90°depending on what camera sensor is being evaluated (Figure 7.16 (a)) or (b)). Take a picture of the object with the same camera used in Step 1.

We can then compare the *distance to object* result of each camera sensor with the known physical distance d between the camera and object (see equation (7.1)). If one of the camera sensor *distance to object* results is not within expected tolerances, then this sensor is 'tagged' as being faulty. If none of the camera sensor

distance to object results are within expected tolerances, then the stereo camera vision device will be declared as fully disabled and not capable of detecting objects within its path. No adjustment can be made for this scenario.

7.6.7 Adjustment (compensating for the stereo camera fault)

If during the Analysis evaluation, it is established that there is at least one working camera sensor, we can implement an *adjustment policy* to compensate for the other faulty sensor (see Algorithm 8). The *compensation* strategy is similar to the strategy employed to discover what stereo camera sensor was operating correctly during the analysis stage - (See Figure 7.16). The *adjustment policy* implemented by the Autonomic Manager will then send instructions to the *operating policy* in the System Manager, which in turn will update the System Manager (Task Module).

ALGORITHM 8: Stereo Vision Camera Fault Adjustment

Input: Enter the identity of the working camera sensor SL (left camera) or SR (right camera), enter the Baseline Value of Stereo Camera.
Output: Using the one camera, process stored image with current image to establish the distance value of an object = *ObjectDistance*

```
if (cameraEnabled = SL) then
    takeImageCameraSensor(SL);
    storeImageDataForCamera(SL);
    rotateRobotCommand(-90);
    moveRobotCommandDistance(BaselineValue);
    rotateRobotCommand(90);
    takeImageCameraSensor(SL);
    ObjectDistance = performImageProcessing(currentImage, storedImage);
end
if (cameraEnabled = SR) then
    takeImageCameraSensor(SR);
    storeImageDataForCamera(SR);
    rotateRobotCommand(90);
    moveRobotCommandDistance(BaselineValue);
    rotateRobotCommand(-90);
    takeImageCameraSensor(SR);
    ObjectDistance = performImageProcessing(currentImage, storedImage);
end
```

7.6.8 Conclusions (compensating for the stereo camera fault)

This case study shows that even with one damaged stereo camera sensor, it is still possible to locate an object using Stereo Vision processing with the assistance of suitable adjustments via the AIFH architecture. The AIFH architecture can be em-

ployed to evaluate mobile robot hardware components if there are specialized policies available to the Autonomic Manager. The *Awareness* process can be used to establish if there is a possible fault within the Stereo Camera component. We can then utilize the *Analysis* procedure to evaluate the extent of the fault. Finally, if there is one fully functional camera sensor available, we can then use an *Adjustment* algorithm to compensate for the fault.

As with any compensation procedure, there will be an effect on how well the mobile robot performs its task. With the Stereo Vision Camera Fault Adjustment algorithm, detecting an object will take a greater amount of time and processing. For short term tasks, this may not be an issue but for longer scheduled tasks, this could affect resources like power management.

7.7 AIFH Autonomic Architecture Summary

The AIFH architecture was initially developed using principles found in the MAPE-K and IMD architectural models [1] [5]. Further development of the AIFH architecture was achieved by using the research carried out in the case studies performed in Chapter's 4, 5 and 6. The final generic AIFH design is presented in Section 7.5, Figure 7.8. The generic AIFH architecture contains Autonomic Manager (containing the 3 layers, *awareness*, *analysis* and *adjustment*), a Knowledge Base module and a *Self-Healing* module. Autonomic Manager is linked to the System Manager which responsible for executing commands to the mobile robot via the *effectors* and *sensors*. The Autonomic Manager provides a mechanism for detecting faults, analyzing faults and providing policies that can compensate for faults. The 'health checking' mechanism is provided by the *self-healing* module. The 'autonomic intelligence' in the *awareness* layer, not only flags component faults but also provides a means of monitoring sensor data and reporting to the User/Mission Control, if a component's behaviour may indicate an impending fault. Within the *analysis* layer, there are policies that can adjust *tolerance* thresholds. These policies are important as an over sensitive tolerance value may lead to faults being reported continuously within the *Reaction Loop* (between the Awareness and Analysis layers), and there-

fore creating an infinite fault state.

In Section 7.6, we employed a case study (Stereo Vision Camera) and applied the AIFH architectural model to that study. The purpose was to demonstrate that the AIFH autonomic architecture can apply to other component fault scenarios that can occur within a mobile robot. This is the first case study where the AIFH architecture is used to map out an Autonomic solution.

This architecture attempts to integrate the autonomic principles of *self-healing*, *self-analyzing*, *self-aware*, *self-optimizing*, as described in research by [47] [123] [124]. The goal of the generic AIFH architecture, is to provide an autonomic solution that can be implemented for any mobile robot type that provides component fault handling without human intervention. The types of parameters associated with a functioning robot that an autonomic manager can use in order to establish *awareness*, *analysis* and *adjustment* for operational problems are virtually limitless. Robot developers need to consider these types of problems so that robots in the future will become better adapted to *fault* handling. Consideration needs to be given to the amount of processing time afforded to the Autonomic Manager. Each sensor on a mobile robot produces a sizable quantity of data and if this data is consumed by the Autonomic Manager (and not regulated), then the System could be overwhelmed.

7.7.0.1 AIFH versus MAPE-K and IMD

Why would a developer/researcher consider using the AIFH architecture rather than MAPE-K or IMD?

In this work, research was conducted on the MAPE-K (Autonomic Computing) and IMD (Robotics) models to adopt key features from those architectures, to formulate a hybrid generic architecture (Autonomic Robotics) that specifically focuses on mobile robot fault handling. The MAPE-K design offers a single feed-back loop that monitors for faults whereas the AIFH offers a dual feed-back loop (reactive and proactive) - this allows, not only to react quickly to fault situations but also to investigate sensor data and look for *downward* trends in component behaviors. The IMD model can react quickly to a fault but it lacks the *knowledge* over time, to establish if a component is under performing. The MAPE-K the feed-back

loop is one-way (Analysis leads to Plan, Plan to Execute etc.); the AIFH (two-way feed-back loop), can make a decision within the Analysis Layer to return back to the Awareness Layer (during its fault analysis), and alert the Awareness Layer that its fault detection process is over-sensitive and needs re-adjusting. In the MAPE-K, the *Execute* process simply carries out the polices from the *Plan* process without question, however, in the AIFH model, the Adjustment Layer takes the place of both MAPE-K (*Plan* and *Execute*), in regard to decision making and execution of compensation policies. In comparison to the IMD model, the Adjustment Layer has a direct route to the *effectors*, to implement policy changes. If the IMD (Reflection Layer), is used to formulate a policy, it must traverse 3 layers before it can communicate with the *effectors*.

Chapter 8

Conclusions and Future Work

This chapter outlines the conclusions and future work of the research presented in this thesis. Section 8.1 discusses a summary of the research and its outcomes. Section 8.2 presents the conclusions of the research. Finally, Section 8.3 presents the possibilities for future development of the research conducted within this thesis.

8.1 Overall Summary

The work presented in this thesis highlights the importance of how autonomic architecture can play a role in handling component faults within a mobile robot. It argues that from the original *autonomic* model presented by IBM [1] and robotics model (IMD), presented in [5], that there is scope to expand on this architecture and use an alternate model (AIFH), that focuses on *self-awareness*, *self-analysis* and *self-healing*. It argues that a *generic* autonomic architecture can be employed to handle a variety of component faults that can be applied to any type of mobile robot. The proposed *generic* architecture (AIFH), was developed over time using the SDLC methodology which was applied to a series of case studies. Each case study contributed to identifying *design patterns* which in-turn supplied the tools to build the *generic* architecture. After the case studies where completed, the resulting insight provided the means to develop a fully-fledged *generic* autonomic architecture (AIFH), for handling faults in mobile robots. To evaluate the AIFH architecture, a further case study was presented that introduced another component *fault* scenario. The AIFH architecture was used to identify the fault (awareness),

process the fault (analysis) and compensate for the fault (adjustment).

- *Chapter 2* introduced the origins of autonomic computing and why it was relevant to the current state of computer systems. This chapter then discussed the *autonomic manager* and the properties of the MAPE-K model. The IMD model was then investigated and how it could be integrated with the MAPE-K model. This chapter also discussed Organic computing and how it differs from autonomic computing. Finally, this chapter explains how authors in this field had incorporated autonomic computing principles into their research and how they used the MAPE-K architecture as a base to create their own autonomic model.

- *Chapter 3* described the *goals* put forward for this thesis and what research methods would be employed to achieve the specified *goals*. This chapter also discussed the concept of a *generic architecture* and how 'awareness' (in autonomic terms), plays a significant role in fault detection in mobile robots. Finally, this chapter explained the use of SDLC as a methodology and how it was applied to each case study in the thesis.

- *Chapter 4* presented a case study describing how autonomic concepts can be used to handle a 'wheel alignment' fault in a mobile robot. This chapter also described the early development of the *generic autonomic* architecture for detecting faults in mobile robots. The concept of 'awareness' was explored in how a fault can be detected or, by examining current and historical data, a fault can be anticipated. Finally, this Chapter presented how an *adjustment* strategy can be employed to compensate for the fault.

- *Chapter 5* presented a case study that examined the effect faulty sonar sensors would have on a mobile robot being able to detect an object. In this Chapter, the concepts of *Awareness*, *Analysis* and *Adjustment* where used to detect, process and compensate for a sonar sensor fault. This Chapter also highlighted the importance of using a *Knowledge Base* for manipulating *Tolerance* values, analyzing *Historical* data and making the use of *Policies* to

perform analysis. Finally, this Chapter concluded with testing and evaluating the *adjustment* policy, in order to prove the mobile robot could still detect objects even with faulty sonar sensors.

- *Chapter 6* presented a case study to describe how battery degradation can affect the performance of a mobile robot. Using a simulated battery model, analysis was performed on how a mobile robot performs with (1), a fully charged battery and (2), a battery exposed to degradation. In this Chapter, the use of *self-awareness*, *self-analysis* and *self-healing* concepts (to handle the battery fault), aided the further development of the *generic autonomic* architecture. Finally, this Chapter described how the *adjustment* policy (algorithm), improved the performance of the mobile robot while using a degraded battery.

- *Chapter 7* describes the fully-fledged generic autonomic architecture for handling faults in mobile robots (AIFH). This Chapter presented a 'high-level' overview of the AIFH architecture: this includes

  (1) - the role played by the System Manager.

  (2) - the role played by the Autonomic Manager.

  (3) - how both the System Manager and Autonomic Manager interact.

  The Chapter then describes the elements found within the Autonomic Manager. These elements include the *self-healing* module and the *knowledge base* module. Further description of the *knowledge base* module, explains the roles that *polices*, *tolerance values* and *historical data* play in handling 'fault' data. The Chapter then describes the 'low-level' detail of the AIFH architecture. The 'low-level' architecture shows how the Reactive and Proactive feedback loops transport data through each layer (awareness, analysis and adjustment). Further description of each layer shows how 'fault' data is manipulated and how each layer interacts with the *knowledge base* module. Finally, this Chapter concludes with describing how a further case study is presented to demonstrate the AIFH generic architecture and how it handles a new fault scenario.

8.2 Conclusions

Although autonomic computing has been around since the beginning of the millennium, it is still a relatively new concept. The research in this thesis has offered an opportunity to investigate an area of computing science that is still evolving and which possess more questions than answers. Fault handling in mobile robots is not a new concept and much research has been conducted over the years using *fault tolerance* techniques. However, the nature of the autonomic computing offers researchers a fresh approach to dealing with faults in mobile robots.

In this thesis, one of the main goals was to take existing 'autonomic and 'intelligent machine' models and combine them to create a new generic architecture. This goal was attained by developing case studies that centred around component faults within a mobile robot. As each case study progressed, 'design patterns' where identified that had shared common elements within the case studies. These common 'elements' where then grouped together to form 3 *layers* within the generic autonomic architecture (AAA - *Awareness*, *Analysis* and *Adjustment*). Building on design, evaluation and experience, each case study provided further structural content to the generic architecture. When the case study process was completed, all the insights gathered from this *process*, formed the design and implementation of the final generic autonomic architecture.

The question is - how can this generic autonomic architecture (AIFH), make a difference to the current research carried out in this area? The AIFH architecture has been designed so that multiple types of robots (with varied sensors), can be *self-managed*. Through developing the AIFH architecture, the research in this thesis identified *autonomic* 'awareness' as one of its goals. This 'awareness' not only monitors incoming data but uses *intelligent monitoring*. If a fault occurs within a component and it becomes *disabled*, then a basic standard of monitoring would be sufficient to alert the system to the fault. However, a component that is exhibiting slight behavioral changes but is still operating within tolerances, is more difficult to implicate. Using the autonomic *knowledge base* in the AIFH architecture, real-time and historical data can be periodically investigated to check for any anomalies

within a component. If anomalies are discovered in a component, then a report can be sent to indicate an impending fault for that particular component.

The development of the generic architecture highlighted limitations in the research in terms of the use of third-party robotic hardware. When attempting to compensate for component faults, not all access is afforded the developer to manipulate parameters within these components. This can restrict the developer or researcher's ability to fully-restore the robot's functionality if a fault has occurred. There are of course certain security implications if all access to systems is unrestricted, so there may have be a requirement to restricted access. However, organizations like NASA, can allow 'all access' for their own developers without compromising security.

In this Thesis, a generic autonomic fault architecture is proposed in order to address deficiencies in identifying not only current faults but more importantly, impending faults. Detailed analyses of real-time and historical data can indicate a downward trend in a components behavior which can lead to early detection of an impending fault. However, there are limitations, like handling multiple fault scenarios and the amount of band-width that can be afforded to the Autonomic Manager to process fault data.

8.3 Future Work

Although the research in this thesis answered the questions that where proposed in Chapter 3, other questions have emerged. There is also scope for further research in adding functionality to the generic architecture. The following is a summary of possible directions for this research:

- **Software implementation of the AIFH architecture:** the AIFH architecture has provided a template for autonomic 'fault handling' in mobile robot systems but further investigation is required to develop the software that will 'realise' the AIFH model in practical terms. Although a considerable amount of software was developed during this research programme, (in terms of manipulating the mobile robot and handling component faults in each case study), it would require considerable effort and further research to apply the AIFH

model to an actual 'mobile robot' system.

- **Autonomic Data Filtering:** one of the questions that emerged as the research in this thesis developed (Chapter 7), was how to limit the amount of sensor data afforded to the Autonomic Manager. If the Autonomic Manager was simply allowed to consume all the data from the sensors, then system could be overwhelmed with the amount of processing time required to analyse the data. Further investigation is required in the development of a 'autonomic' filter that can be 'aware' of the limitations of its environment and therefore regulate the amount of data taken from the sensors. This would be an interesting experiment, although complex, in terms of assessing the amount of processing available in the system against the needs of an Autonomic Manager when detecting anomalies within the system components.

- **Multi-Agent Architecture:** the research conducted in this thesis centred on a single managed robotic unit. Future development could focus on implementing multiple robotic units. If a robot using the AIFH architecture had been operating in the field for sometime, it would have accumulated a considerable amount of knowledge. This knowledge would include fault history and polices that where used to process the fault. This knowledge could be shared with other neighbouring robotic units. Future research could look at implementing the AIFH model as a multi-tier or multi-dimensional design.

- **Further case studies:** although a total of four component types within mobile robots where investigated in this research (drive system, sonar sensors, stereo vision cameras and power supply), further research could be conducted on alternative complementary components. These components could include laser sensors, robotic arms, infrared sensors and pan and tilt servos. case studies conducted around 'faults' in these components would broaden knowledge for further enhancement of the AIFH architecture.

Appendix A

Case Study Reference: Wheel Alignment Fault

A.1 Pioneer P3-DX Robot *laser* alignment readings

ID	Test Type	Test Time	Start	End	Distances	Batch	Diff.
44	Alignment	4/21/2015 3:11:30 PM	422	420	250	1001	2
45	Alignment	4/21/2015 3:12:17 PM	418	409	250	1001	9
46	Alignment	4/21/2015 3:13:04 PM	418	427	250	1001	-9
47	Alignment	4/21/2015 3:13:52 PM	417	413	250	1001	4
48	Alignment	4/21/2015 3:14:39 PM	415	410	250	1001	5
49	Alignment	4/21/2015 3:15:26 PM	419	426	250	1001	-7
50	Alignment	4/21/2015 3:16:13 PM	415	437	250	1001	-2
51	Alignment	4/21/2015 3:17:00 PM	414	407	250	1001	7
52	Alignment	4/21/2015 3:17:47 PM	412	418	250	1001	-6
53	Alignment	4/21/2015 3:18:34 PM	418	423	250	1001	-5
							-0.2
							mm

ID	Test Type	Test Time	Start	End	Distance	Batch	Diff.
54	Alignment	4/22/2015 10:22:04 AM	420	428	250	1002	-8
55	Alignment	4/22/2015 10:22:51 AM	413	426	250	1002	-13
56	Alignment	4/22/2015 10:23:38 AM	418	431	250	1002	-13
57	Alignment	4/22/2015 10:24:25 AM	415	426	250	1002	-11
58	Alignment	4/22/2015 10:25:12 AM	416	430	250	1002	-14
59	Alignment	4/22/2015 10:25:59 AM	416	420	250	1002	-4
60	Alignment	4/22/2015 10:26:46 AM	418	431	250	1002	-13
61	Alignment	4/22/2015 10:27:33 AM	412	410	250	1002	2
62	Alignment	4/22/2015 10:28:20 AM	413	419	250	1002	-6
63	Alignment	4/22/2015 10:29:07 AM	412	418	250	1002	-6
							-8.6
							mm

ID	Test Type	Test Time	Start	End	Distance	Batch	Diff.
64	Alignment	4/22/2015 11:30:52 AM	425	422	250	1003	3
65	Alignment	4/22/2015 11:31:40 AM	417	438	250	1003	-21
66	Alignment	4/22/2015 11:32:27 AM	417	429	250	1003	-12
67	Alignment	4/22/2015 11:33:14 AM	415	425	250	1003	-10
68	Alignment	4/22/2015 11:34:01 AM	415	411	250	1003	4
69	Alignment	4/22/2015 11:34:48 AM	411	421	250	1003	-10
70	Alignment	4/22/2015 11:35:35 AM	417	408	250	1003	9
71	Alignment	4/22/2015 11:36:22 AM	415	426	250	1003	-11
72	Alignment	4/22/2015 11:37:09 AM	416	420	250	1003	-4
73	Alignment	4/22/2015 11:37:56 AM	416	437	250	1003	-21
							-7.3
							mm

ID	Test Type	Test Time	Start	End	Distance	Batch	Diff.
77	Alignment	4/22/2015 2:35:56 PM	415	390	250	1004	-25
78	Alignment	4/22/2015 2:36:43 PM	422	392	250	1004	-30
79	Alignment	4/22/2015 2:37:30 PM	420	387	250	1004	-33
80	Alignment	4/22/2015 2:38:17 PM	422	378	250	1004	-44
81	Alignment	4/22/2015 2:39:04 PM	421	376	250	1004	-45
82	Alignment	4/22/2015 2:39:51 PM	413	393	250	1004	-20
83	Alignment	4/22/2015 2:40:38 PM	415	382	250	1004	-33
84	Alignment	4/22/2015 2:41:25 PM	411	372	250	1004	-39
85	Alignment	4/22/2015 2:42:13 PM	415	393	250	1004	-22
86	Alignment	4/22/2015 2:43:00 PM	410	351	250	1004	-59
							-35
							mm

ID	Test Type	Test Time	Start	End	Distance	Batch	Diff.
87	Alignment	4/22/2015 5:21:35 PM	423	389	250	1005	-34
88	Alignment	4/22/2015 5:22:23 PM	420	382	250	1005	-38
89	Alignment	4/22/2015 5:23:10 PM	421	369	250	1005	-52
90	Alignment	4/22/2015 5:23:57 PM	418	375	250	1005	-43
91	Alignment	4/22/2015 5:24:44 PM	417	370	250	1005	-47
92	Alignment	4/22/2015 5:25:31 PM	415	370	250	1005	-45
93	Alignment	4/22/2015 5:26:18 PM	414	360	250	1005	-54
94	Alignment	4/22/2015 5:27:05 PM	411	356	250	1005	-55
95	Alignment	4/22/2015 5:27:52 PM	410	341	250	1005	-69
96	Alignment	4/22/2015 5:28:39 PM	412	355	250	1005	-57
							-49.4
							mm

ID	Test Type	Test Time	Start	End	Distance	Batch	Diff.
97	Alignment	4/22/2015 5:52:44 PM	419	378	250	1006	-28
98	Alignment	4/22/2015 5:53:31 PM	420	392	250	1006	-49
99	Alignment	4/22/2015 5:54:18 PM	418	369	250	1006	-45
100	Alignment	4/22/2015 5:55:05 PM	414	369	250	1006	-55
101	Alignment	4/22/2015 5:55:52 PM	409	354	250	1006	-64
102	Alignment	4/22/2015 5:56:40 PM	416	351	250	1006	-54
103	Alignment	4/22/2015 5:57:27 PM	415	361	250	1006	-20
104	Alignment	4/22/2015 5:58:14 PM	412	392	250	1006	-51
105	Alignment	4/22/2015 5:59:01 PM	410	359	250	1006	-69
106	Alignment	4/22/2015 5:59:48 PM	413	344	250	1006	-48
							-48.3
							mm

Appendix B

Case Study Reference: Sonar Sensor Fault

B.1 Pioneer P3-DX Robot - sonar sensor fault states and compensation rotation values

Using positive rotation as first priority

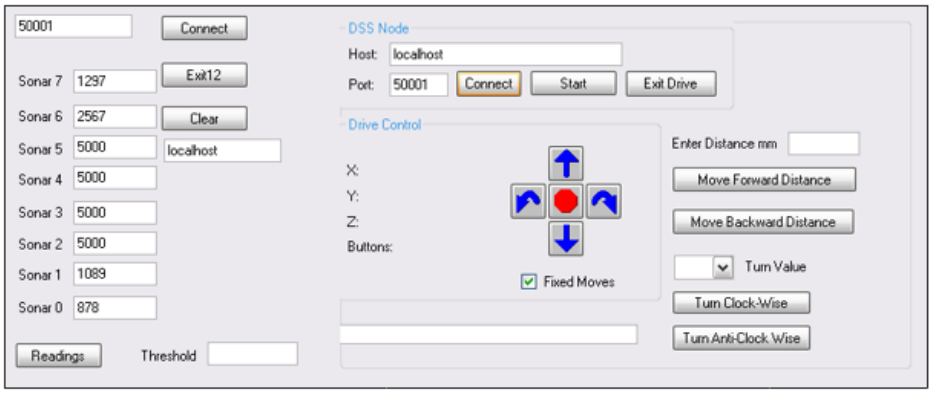
~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~		
0	0	0	0	0	1		R + 20°
0	0	0	0	1	0		R + 20°
0	0	0	0	1	1		R + 40°
0	0	0	1	0	0		R + 20°
0	0	0	1	0	1		R + 20°
0	0	0	1	1	0		R + 40°
0	0	0	1	1	1		R + 60°
0	0	1	0	0	0		R + 20°
0	0	1	0	0	1		R + 20°
0	0	1	0	1	0		R + 20°
0	0	1	0	1	1		R + 20° and R + 20°
0	0	1	1	0	0		R + 40°
0	0	1	1	0	1		R + 40° and R - 20°
0	0	1	1	1	0		R + 40° and R - 60°
0	0	1	1	1	1		R + 40° and R + 40°
0	1	0	0	0	0		R + 20°
0	1	0	0	0	1		R + 20°
0	1	0	0	1	0		R + 20°
0	1	0	0	1	1		R + 20° and R + 20°
0	1	0	1	0	0		R + 20°
0	1	0	1	0	1		R + 20°
0	1	0	1	1	0		R + 20° and R - 40°
0	1	0	1	1	1		R + 20° and R + 20 ° + R + 20 °
0	1	1	0	0	0		R - 40°
0	1	1	0	0	1		R + 20° and R - 40°
0	1	1	0	1	0		R + 20° and R - 40°
0	1	1	0	1	1		R + 20° and R + 20°
0	1	1	1	0	0		R + 20° and R - 60°
0	1	1	1	0	1		R + 20° and R + 20° and R + 20°
0	1	1	1	1	0		R + 20° and R + 20° and R - 60° and R - 20°
0	1	1	1	1	1		R + 20° and R + 20° and R + 20° R + 20° and R + 20°

Using negative rotation as first priority

1	0	0	0	0	0	R - 20°
1	0	0	0	0	1	R - 20° and R + 40°
1	0	0	0	1	0	R - 20°
1	0	0	0	1	1	R - 20° and R + 60°
1	0	0	1	0	0	R - 20°
1	0	0	1	0	1	R - 20° and R + 40°
1	0	0	1	1	0	R - 20° and R + 60°
1	0	0	1	1	1	R - 20° and R + 40° and R +20°
1	0	1	0	0	0	R - 20°
1	0	1	0	0	1	R - 20° and R + 60°
1	0	1	0	1	0	R - 20°
1	0	1	0	1	1	R - 20° and R + 60°
1	0	1	1	0	0	R - 20° and R -20°
1	0	1	1	0	1	R - 20° and R + 40°
1	0	1	1	1	0	R - 20° and R - 20° and - 20°
1	0	1	1	1	1	R - 20° and R + 40° and +20° (x3)
1	1	0	0	0	0	R - 40°
1	1	0	0	0	1	R - 40° and R + 60°
1	1	0	0	1	0	R - 20° and R - 20° and - 20°
1	1	0	0	1	1	R - 40° and R + 80°
1	1	0	1	0	0	R - 20° and R - 20°
1	1	0	1	0	1	R - 20° and R - 20° and + 60°
1	1	0	1	1	0	R - 20° and R -20°
1	1	0	1	1	1	R -20° and R -20° and R +60° and R +20° and R +20°
1	1	1	0	0	0	R - 60°
1	1	1	0	0	1	R - 40° and R -20° and R + 80°
1	1	1	0	1	0	R - 20° and R -20° and R -20 °
1	1	1	0	1	1	R -20° and R -20° and R -20 ° and R + 80° and R -20°
1	1	1	1	0	0	R -40° and R -40°
1	1	1	1	0	1	R -20° and R -20° and R -20° and R -20° + and R -100
1	1	1	1	1	0	R - 20° and R -20° and R -20° R -20° and R -20°
~~1~~	~~1~~	~~1~~	~~1~~	~~1~~	~~1~~	

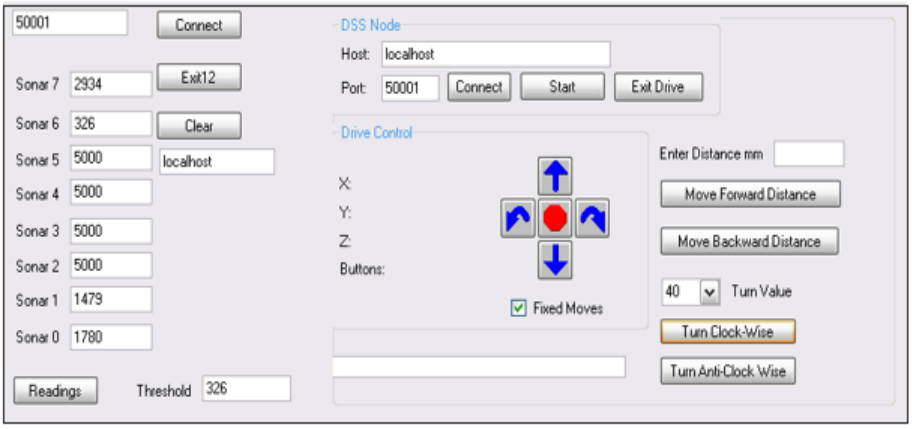
B.2 Sonar Sensor Fault - Compensation experiment

Experiment 1: (software written by author) - Sonar Transducers 2, 3, 4 and 5 are disabled. The remaining sonars have readings greater than the object detection threshold (set at 350 mm).



Using the rotation formula, the lowest position of a disabled transducer is -10 (Sonar 4), the next available transducer is 50 (Sonar 6). If the signs are equal then simply subtract 10 from 50. The rotation required is +40°

Experiment 2:



The reading for Sonar 6 shows 326, which is below the Threshold and would indicate that an object has been found. Likewise, the opposite enabled sensors could be used for detection. In this case the robot would have been rotated anti-clock wise (-40°).

Bibliography

[1] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, pp. 41–50, Jan 2003.

[2] E. Rutten, N. Marchand, and D. Simon, "Feedback Control as MAPE-K loop in Autonomic Computing," Research Report RR-8827, INRIA Sophia Antipolis - Méditerranée ; INRIA Grenoble - Rhône-Alpes, Dec. 2015. draft soumis à LNCS.

[3] "Artificial intelligence: A new synthesis," in *Artificial Intelligence: A New Synthesis* (N. J. Nilsson, ed.), p. iv, Oxford: Morgan Kaufmann, 1998.

[4] A. Sloman, "Evolvable architectures for human-like minds," in *Affective Minds*, pp. 169–181, Elsevier, 2000.

[5] D. A. Norman, A. Ortony, and D. M. Russell, "Affect and machine design: Lessons for the development of autonomous machines," *IBM Systems Journal*, vol. 42, no. 1, pp. 38–44, 2003.

[6] H. Shualib, R. Anthony, and M. Pelc, "A framework for certifying autonomic computing systems," in *The Seventh international Conference on Autonomic and Autonomous Systems*, pp. 122–127, 01 2011.

[7] J. Branke, M. Mnif, C. Mller-Schloer, H. Prothmann, U. Richter, F. Rochner, and H. Schmeck, "Organic computing addressing complexity by controlled self-organization," in *Conference: Leveraging Applications of Formal Methods, Second International Symposium*, pp. 185–191, 11 2006.

[8] B. Jakimovski, M. Litza, F. Mosch, and A. El-Sayed-Auf, "Development of an organic computing architecture of robot control," in *Informatik 2006 Workshop on Organic Computing*, vol. 1, pp. 145–152, January 2006.

[9] C. C. Insaurralde, "Autonomic management capabilities for robotics and automation," vol. 1, no. 1, pp. 518–523, 2013.

[10] A. Hernando, R. Sanz, and R. Calinescu, "A model-based approach to the autonomic management of mobile robot resources," in *ADAPTIVE 2010, The Second International Conference on Adaptive and Self-Adaptive Systems and Applications*, pp. 33–39, November 2010.

[11] C. Pahl and P. Jamshidi, "Software architecture for the cloud – a roadmap towards control-theoretic, model-based cloud architecture," in *Software Architecture*, pp. 212–220, Springer International Publishing, 2015.

[12] T. A. Nguyen, M. Aiello, T. Yonezawa, and K. Tei, "A self-healing framework for online sensor data," in *2015 IEEE International Conference on Autonomic Computing*, pp. 295–300, July 2015.

[13] M. Viroli, A. Bucchiarone, D. Pianini, and J. Beal, "Combining self-organisation and autonomic computing in cass with aggregate-mape," in *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pp. 186–191, Sept 2016.

[14] S. Karapinar, D. Altan, and S. Sariel-Talay, "A robust planning framework for cognitive robots," in *In Proceedings of the AAAI-12 workshop on cognitive robotics (CogRob)*, pp. 102–108, 2012.

[15] M. Doran, R. Sterritt, and G. Wilkie, "Autonomic wheel alignment for mobile robots," in *11th IEEE International Conference and Workshops on the Engineering of Autonomic and Autonomous*, (Laurel, Washington, USA), p. 6, September 2014.

[16] Y. B. LTD, "Np series np7.5-12 data sheet," 2008. [Online]. Available: http://www.yuasabatteries.com/np-industrial-literature.php. Last accessed, December 10th, 2017.

[17] Y. Mei, Y.-H. Lu, Y. C. Hu, and C. S. G. Lee, "A case study of mobile robot's energy consumption and conservation techniques," in *ICAR '05. Proceedings., 12th International Conference on Advanced Robotics, 2005.*, pp. 492–497, July 2005.

[18] B. Kuipers, E. A. Feigenbaum, Edward, P. E. Hart, and N. Nilsson, "Shakey: From conception to history," *Ai Magazine*, vol. 38, pp. 88–103, 03 2017.

[19] Robotics-Industrial-Association, "Industrial mobile robot safety standards on the forefront," 2017. [Online]. Available: https://www.robotics.org/content-detail.cfm/Industrial-Robotics-Industry-Insights/Industrial-Mobile-Robot-Safety-Standards-on-the-Forefront/contentid/6710. Last accessed, July 20th, 2019.

[20] A. C. Leite, B. Schafer, and M. L. de Oliveira e Souza, "Fault-tolerant control strategy for steering failures in wheeled planetary rovers," *Journal of Robotics*, vol. 2012, pp. 1–15, 2012.

[21] C. Leger, A. Trebi-ollennu, J. Wright, S. Maxwell, R. Bonitz, J. Biesiadecki, F. Hartman, B. Cooper, E. Baumgartner, and M. Maimone, "Mars exploration rover surface operations: Driving spirit at gusev crater," vol. 2, pp. 1815–1822, 11 2005.

[22] NASA, "Six things about opportunity's recovery efforts," 2018. [Online]. Available: https://mars.nasa.gov/news/8360/six-things-about-opportunitys-recovery-efforts/?site=insight. Last accessed, July 20th, 2019.

[23] *A Dictionary of English (12th Edition)*. HarperCollins Publishers, 2014. Collins COBUILD.

[24] R. Sterritt and M. Hinchey, "Autonomic computing - panacea or poppycock?," in *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, pp. 535–539, April 2005.

[25] M. Parashar and S. Hariri, "Autonomic computing: An overview," in *Unconventional Programming Paradigms*, vol. 3566, (Berlin, Heidelberg), pp. 257–269, Springer Berlin Heidelberg, 2005.

[26] A. G. Ganek and T. A. Corbi, "The dawning of the autonomic computing era," *IBM Systems Journal*, vol. 42, no. 1, pp. 5–18, 2003.

[27] R. Sterritt and M. Hinchey, "Why computer-based systems should be autonomic," in *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, pp. 406–412, April 2005.

[28] W. F. Truszkowski, M. G. Hinchey, J. L. Rash, and C. A. Rouff, "Autonomous and autonomic systems: a paradigm for future space exploration missions," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 36, pp. 279–291, May 2006.

[29] D. Crestani, K. Godary-Dejean, and L. Lapierre, "Enhancing fault tolerance of autonomous mobile robots," *Robotics and Autonomous Systems*, vol. 68, pp. 140 – 155, 2015.

[30] G. K. Fourlas, G. C. Karras, and K. J. Kyriakopoulos, "Sensors fault diagnosis in autonomous mobile robots using observer based technique," in *2015 International Conference on Control, Automation and Robotics*, pp. 49–54, May 2015.

[31] O. Zweigle, B. Keil, M. Wittlinger, K. Haussermann, and P. Levi, *Recognizing Hardware Faults on Mobile Robots Using Situation Analysis Techniques*, pp. 397–409. Springer, 2013.

[32] A. Almeida, J. Briot, S. Aknine, Z. Guessoum, and O. Marin, "Towards autonomic fault-tolerant multi-agent systems," in *In The 2nd Latin American*

Autonomic Computing Symposium (LAACS2007), Petropolis, RJ, Brsil, 01 2007.

[33] A. Paz and H. Arboleda, "A model to guide dynamic adaptation planning in self-adaptive systems," *Electronic Notes in Theoretical Computer Science*, vol. 321, pp. 67 – 88, 2016. CLEI 2015, the XLI Latin American Computing Conference.

[34] S. Seshachala, "Cloud computing architecture: Front end and back end," 2015. [Online]. Available: https://cloudacademy.com/blog/cloud-computing-architecture-an-overview. Last accessed 03 June 2018.

[35] O. Selma, S. Boulehouache, and S. Mazouzi, "A survey of uncertainties in mape-k control loop," in *Conference: International Conference on Advanced Technologies, Computer Engineering and Science (ICATCES18), May 11-13, 2018., At Safranbolu, Turkey*, 05 2018.

[36] D. Weyns, S. Malek, and J. Andersson, "Forms: A formal reference model for self-adaptation," in *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, (New York, NY, USA), pp. 205–214, ACM, 2010.

[37] R. Sterritt, G. Wilkie, G. Brady, C. Saunders, and M. Doran, "Autonomic robotics for future space missions," in *13th Symposium on Advanced Space Technologies in Robotics and Automation (ASTRA 2015)*, p. 7, May 2015.

[38] M. Doran, R. Sterritt, and G. Wilkie, "Autonomic self-adaptive robot wheel alignment," in *Proceedings of the 7th International Conference on Autonomic Computing*, pp. 27–33, IARIA, March 2016.

[39] M. Doran, R. Sterritt, and G. Wilkie, "Autonomic sonar sensor fault manager for mobile robots," in *19th International Conference on Autonomic Computing and Computer Engineering, London, UK. WASET*, vol. 11, pp. 621–628, May 2017.

[40] M. Doran, R. Sterritt, and G. Wilkie, "Autonomic management for mobile robot battery degradation," in *20th International Conference on Autonomic Computing and Computer Engineering, London, UK. WASET*, vol. 12, pp. 273–279, May 2018.

[41] P. Horn, "Autonomic computing: Ibms perspective on the state of information technology," 2001. [Online]. Available: http://www.research.ibm.com/autonomic/manifesto/ autonomic computing.pdf. Last accessed, March 20th, 2016.

[42] IBM, "Ibm: Autonomic computing: The solution," 2001. [Online]. Available: http://www.research. ibm.com/autonomic/overview/solution.html. Last accessed, May 5th, 2016.

[43] R. Murch, "Introducing autonomic computing," 2004. [Online]. Available: http://www.informit.com/articles/article.aspx?p=333858seqNum=2. Last accessed, May 5th, 2018.

[44] S. Dobson, R. Sterritt, P. Nixon, and M. Hinchey, "Fulfilling the vision of autonomic computing," *Computer*, vol. 43, pp. 35–41, Jan 2010.

[45] A. Banafa, "what is autonomic computing," 2016. [Online]. Available: https://www.bbvaopenmind.com/en/what-is-autonomic-computing/. Last accessed May 7th, 2018.

[46] R. Sterritt, M. Parashar, H. Tianfield, and R. Unland, "A concise introduction to autonomic computing," *Adv. Eng. Inform.*, vol. 19, pp. 181–187, July 2005.

[47] "An architectural blueprint for autonomic computing," June 2005. [Online]. Available: https://www-03.ibm.com/autonomic/pdfs/AC-20Blueprint-20White-20Paper-20V7.pdf. Last Accessed, March 9th, 2018.

[48] M. Hamblem, "Ibm launches self-healing tools," 2005. [Online]. Available: https://www.computerworld.com/article/2560448/enterprise-

applications/ibm-launches-self-healing-tools.html. Last accessed, May 20th, 2018.

[49] P. Lalanda, J. A. McCann, and A. Diaconescu, *Autonomic Computing - Principles, Design and Implementation*. Undergraduate Topics in Computer Science, London: Springer-Verlag London, 2013.

[50] B. Miller, "The autonomic computing edge: The role of knowledge in autonomic systems," 2005. [Online]. Available: https://www.ibm.com/developerworks/library/ac-edge6/index.html. Last accessed, May 10th, 2018.

[51] E. Eryilmaz, F. Trollmann, and S. Albayrak, "Conceptual application of the mape-k feedback loop to opportunistic sensing," in *2015 Sensor Data Fusion: Trends, Solutions, Applications (SDF)*, pp. 1–6, Oct 2015.

[52] B. A. Caprarescu and D. Petcu, "A self-organizing feedback loop for autonomic computing," in *2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, pp. 126–131, Nov 2009.

[53] A. Farahani, G. Cabri, and E. Nazemi, "Self-* properties in collective adaptive systems," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*, UbiComp '16, (New York, NY, USA), pp. 1309–1314, ACM, 2016.

[54] J. O. Kephart and W. E. Walsh, "An artificial intelligence perspective on autonomic computing policies," in *Proceedings. Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004. POLICY 2004.*, pp. 3–12, June 2004.

[55] H. A. Simon, "Motivational and emotional controls of cognition," *Psychological review*, vol. 74, pp. 29–39, 02 1967.

[56] C. Rouff, M. Hinchey, J. Rash, W. Truszkowski, and R. Sterritt, "Autonomicity of nasa missions," in *Second International Conference on Autonomic Computing (ICAC'05)*, pp. 387–388, June 2005.

[57] K. M. Lee, K. I. Kim, and J. Yoo, "Autonomicity levels and requirements for automated machine learning," in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, RACS '17, (New York, NY, USA), pp. 46–48, ACM, 2017.

[58] Microsoft, "Windows system resource manager," vol. 1, Autgust 2003. Microsoft White Paper.

[59] C. A. Rouff, M. G. Hinchey, J. L. Rash, W. F. Truszkowski, and R. Sterritt, "Towards autonomic management of nasa missions," in *11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, vol. 2, pp. 473–477, July 2005.

[60] A. Kumar, A. Tayal, K. R. K. Senthil, and B. S. Bindhumadhava, "Multi-agent autonomic architecture based agent-web services," in *2008 16th International Conference on Advanced Computing and Communications*, pp. 329–333, Dec 2008.

[61] R. Buyya, R. N. Calheiros, and X. Li, "Autonomic cloud computing: Open challenges and architectural elements," in *2012 Third International Conference on Emerging Applications of Information Technology*, pp. 3–10, Nov 2012.

[62] J. Kinsella, "why the future is cloud autonomics," 2015. [Online]. Available: https://www.cloudcomputing-news.net/news/2015/may/14/why-future-cloud-autonomics/. Last accessed, June 1st, 2018.

[63] X. Long, X. Gong, X. Que, W. Wang, B. Liu, S. Jiang, and N. Kong, "Autonomic networking: Architecture design and standardization," *IEEE Internet Computing*, vol. 21, no. 5, pp. 48–53, 2017.

[64] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli, "A survey of autonomic communications," *ACM Trans. Auton. Adapt. Syst.*, vol. 1, pp. 223–259, Dec. 2006.

[65] N. Hussain, H. H. Wang, and C. Buckingham, "Policy based generic autonomic adapter for a context-aware social-collaborative system," in *2018 International Conference on Intelligent Systems and Computer Vision (ISCV)*, pp. 1–9, April 2018.

[66] R. Strader, "Why we're working with autonomic to create a platform that can power future cities," 2018. [Online]. Available: https://medium.com/cityoftomorrow/why-were-working-with-autonomic-to-create-a-platform-that-can-power-future-cities-96700c2824e6. Last accessed, June 2nd, 2018.

[67] Car-Magazine, "Autonomous car levels, driver-less technology levels explained," 2018. [Online]. Available: https://www.carmagazine.co.uk/car-news/tech/autonomous-car-levels-different-driverless-technology-levels-explained. Last accessed 07 July 2017.

[68] C. Müller-Schloer, C. von der Malsburg, and R. P. Würt, "Organic computing," *Informatik-Spektrum*, vol. 27, pp. 332–336, Aug 2004.

[69] C. von der Malsburg, *The Organic Future of Information Technology*, vol. 2008, pp. 7–24. Springer-Verlag Berlin Heidelberg, 10 2008.

[70] L. David, "How wheel damage affects mars rover curiosity's mission," 2014. [Online]. Available: https://www.space.com/26472-mars-rover-curiosity-wheel-damage.html. Last accessed, February 9th, 2018.

[71] N. A. Melchior and W. D. Smart, "Autonomic systems for mobile robots," in *International Conference on Autonomic Computing, 2004. Proceedings.*, pp. 280–281, May 2004.

[72] P. Krzyzanowski, "Distributed systems - fault tolerance - dealing with an imperfect world," 2009. [Online]. Available: https://www.cs.rutgers.edu/ pxk/rutgers/notes/content/ft.html. Last accessed, May 20th, 2018.

[73] C. Rouff, J. Rash, and W. Truszkowski, "Overcoming robotic failures through autonomicity," in *Engineering of Autonomic and Autonomous Systems, 2007. EASe '07. Fourth IEEE International Workshop on*, pp. 154–162, March 2007.

[74] M. Visinsky, J. Cavallaro, and I. Walker, "Robotic fault detection and fault tolerance: A survey," *Reliability Engineering and System Safety*, vol. 46, no. 2, pp. 139 – 158, 1994.

[75] Y. Tohma, "Incorporating fault tolerance into an autonomic-computing environment," *IEEE Distributed Systems Online*, vol. 5, pp. 3/1–3/12, Feb 2004.

[76] R. Sterritt and D. Bustard, "Autonomic computing - a means of achieving dependability?," in *10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, 2003. Proceedings.*, pp. 247–251, April 2003.

[77] D. Crestani and K. Godary-Dejean, "Fault Tolerance in Control Architectures for Mobile Robots: Fantasy or Reality?," in *CAR: Control Architectures of Robots*, (Nancy, France), May 2012.

[78] M. Scheutz and J. Kramer, "Reflection and reasoning mechanisms for failure detection and recovery in a distributed robotic architecture for complex robots," in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pp. 3699–3704, April 2007.

[79] E. Khalastchi, M. Kalech, L. Rokach, Y. Shicel, and G. Bodek, "Sensor fault detection and diagnosis for autonomous systems," in *23rd International Workshop on Principles of Diagnosis (DX 2012)*, 2012.

[80] N. Tcholtchev and R. Chaparadza, "Autonomic fault-management and resilience from the perspective of the network operation personnel," in *2010 IEEE Globecom Workshops*, pp. 469–474, Dec 2010.

[81] B. Jakimovski, B. Meyer, and E. Maehle, "Self-reconfiguring hexapod robot oscar using organically inspired approaches and innovative robot leg amputation mechanism," *Institute of Computer Engineering, University of Luebeck, Germany*, January 2009.

[82] J. a. P. Barraca, R. Sadeghi, and R. L. Aguiar, "Collaborative relaying strategies in autonomic management of mobile robotics," *Wirel. Pers. Commun.*, vol. 70, pp. 1077–1096, June 2013.

[83] J. Eger, "The city of the future-the role of telecommunications," March 1994.

[84] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[85] S. G. Tzafestas, "14 - generic systemic and software architectures for mobile robot intelligent control," in *Introduction to Mobile Robot Control* (S. G. Tzafestas, ed.), pp. 589 – 633, Oxford: Elsevier, 2014.

[86] R. Chatila, E. Renaudo, M. Andries, R.-O. Chavez-Garcia, P. Luce-Vayrac, R. Gottstein, R. Alami, A. Clodic, S. Devin, B. Girard, and M. Khamassi, "Toward self-aware robots," *Frontiers in Robotics and AI*, vol. 5, p. 88, 2018.

[87] R. Kwiatkowski and H. Lipson, "Task-agnostic self-modeling machines," *Science Robotics*, vol. 4, no. 26, 2019.

[88] E. N. Skoundrianos and S. G. Tzafestas, "Finding fault - fault diagnosis on the wheels of a mobile robot using local model neural networks," *IEEE Robotics Automation Magazine*, vol. 11, pp. 83–90, Sept 2004.

[89] P. Sundvall and P. Jensfelt, "Fault detection for mobile robots using redundant positioning systems," in *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pp. 3781–3786, May 2006.

[90] G. K. Fourlas, S. Karkanis, G. C. Karras, and K. J. Kyriakopoulos, "Model based actuator fault diagnosis for a mobile robot," in *2014 IEEE International Conference on Industrial Technology (ICIT)*, pp. 79–84, Feb 2014.

[91] O. Hrizi, B. Boussaid, A. Zouinkhi, M. N. Abdelkrim, and C. Aubrun, "Robust adaptive observer based on multi wheeled mobile robot cooperation algorithm," *Automatika*, vol. 57, no. 4, pp. 982–995, 2016.

[92] N.-I. of Open-Schooling, "Phases of system development life cycle," 2005. [Online]. Available: http://oer.nios.ac.in/wiki/index.php/Phases-of-System-Development-Life-Cycle. Last accessed, June 24th, 2018.

[93] Adept-Mobile-Robots, "Pioneer p3-dx mobile robot," 2011. [Online]. Available: http://www.mobilerobots.com/Libraries/Downloads/Pioneer3DX-P3DX-RevA.sflb.ashx. Last accessed, June 17th, 2018.

[94] K. Johns and T. Taylor, *Professional Microsoft Robotics Developer Studio*. Birmingham, UK: Wrox Press Ltd., 2008.

[95] J. S. Cepeda, L. Chaimowicz, and R. Soto, "Exploring microsoft robotics studio as a mechanism for service-oriented robotics," in *2010 Latin American Robotics Symposium and Intelligent Robotics Meeting*, pp. 7–12, Oct 2010.

[96] J. Lawton, "Rass: Resilient autonomic software systems," 2018. [Online]. Available: https://cs.gmu.edu/ menasce/rass/. Last accessed, November 15th, 2018.

[97] E. Khalastchi and M. Kalech, "On fault detection and diagnosis in robotic systems," *ACM Comput. Surv.*, vol. 51, pp. 9:1–9:24, Jan. 2018.

[98] A. Burguera, Y. Cid, and G. Oliver, "Sonar sensor models and their application to mobile robot localization," vol. 9, pp. 10217–43, 12 2009.

[99] E. Tunstel and A. Howard, "Sensing and perception challenges of planetary surface robotics," in *SENSORS, 2002 IEEE*, vol. 2, pp. 1696–1701 vol.2, June 2002.

[100] D. P. Miller, T. Hunt, M. J. Roman, S. Swindell, L. L. Tan, and A. Winterholler, "Experiments with a long-range planetary rover," in *Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2003.

[101] T. Huntsberger, "Fault tolerant action selection for planetary rover control," in *In Proc. Sensor Fusion and Decentralized Control in Robotic Systems, SPIE*, pp. 150–156, 1998.

[102] T. Khler, E. Berghfer, C. Rauch, and F. Kirchner, "Sensor fault detection and compensation in lunar/planetary robot missions using time-series prediction based on machine learning," in *In Acta Futura, ESA Advanced Concepts Team, ESTEC.*, vol. 9, pp. 9–20, May 2014.

[103] M. K. Habib, "Real time mapping and dynamic navigation for mobile robots," *International Journal of Advanced Robotic Systems*, vol. 4, no. 3, p. 35, 2007.

[104] Power-Thru, "Lead acid battery working - lifetime study," 2010. [Online]. Available: http://www.power-thru.com/documents/. Last accessed, March 10th, 2019.

[105] D. C. C. Freitas, M. B. Ketzer, M. R. A. Morais, and A. M. N. Lima, "Lifetime estimation technique for lead-acid batteries," in *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*, pp. 2076–2081, Oct 2016.

[106] American-Scientific, "American scientific: Could battery advances mean better robots?," 2017. [Online]. Available: https//www.scientificamerican.com/article/robot-battery-technology-life-spa/. Last accessed, August 20th, 2018.

[107] L. Nestor, C. Cosmin, T. Hirth, J. G, and F. Dressler, "Robm2: Measurement of battery capacity in mobile robot systems," in *System Software for Persuasive Computing*, pp. 13–18, October 2005.

[108] A. Sweet, G. Gorospe, M. Daigle, J. Celaya, E. Balaban, I. Roychoudhury, and S. Narasimhan, "Demonstration of prognostics-enabled decision making algorithms on a hardware mobile robot test platform," vol. 5, pp. 142–150, August 2014.

[109] Yuasa, "Np serires np7.5-12 data sheet," 2008. [Online]. Available: http://www.yuasabatteries.com. Last accessed, March 10, 2019.

[110] A. G. Ritchie, B. Lakeman, P. Burr, P. Carter, P. N. Barnes, and P. Bowles, *Battery Degradation and Ageing*, pp. 523–527. Boston, MA: Springer US, 2001.

[111] P. Dynamics, "Battery basic," 2011. [Online]. Available: https://www.progressivedyn.com/service/battery-basics/. Last accessed, March 10th, 2019.

[112] Omron, "Pioneer 3 operations manual, version 5," 2007. [Online]. Available: http://www.mobilerobots.com/ResearchRobots/PioneerP3DX.aspx. Last accessed, December 15th, 2017.

[113] D. P. Siewiorek and P. Narasimhan, "Fault-tolerant architectures for space and avionics applications," *First International Forum on Integrated System Health Engineering and Management in Aerospace*, pp. 1–19, November 2005.

[114] L. Yuan, J. S. Dong, J. Sun, and H. A. Basit, "Generic fault tolerant software architecture reasoning and customization," *IEEE Transactions on Reliability*, vol. 55, pp. 421–435, Sep. 2006.

[115] N. T. de Sousa, W. Hasselbring, T. Weber, and D. Kranzlm"uller, "Designing a generic research data infrastructure architecture with continuous soft-

ware engineering," in *Software Engineering Workshops 2018*, vol. Vol-2066 of *CEUR Workshop Proceedings*, pp. 85–88, CEUR-WS.org, March 2018.

[116] A. N. Sylla, M. Louvel, E. Rutten, and G. Delaval, "Design framework for reliable multiple autonomic loops in smart environments," in *2017 International Conference on Cloud and Autonomic Computing (ICCAC)*, pp. 131–142, Sept 2017.

[117] S. S. Laster and A. O. Olatunji, "Autonomic computing: Towards a self-healing system," in *Proceedings of the Spring 2007 American Society for Engineering Education Illinois-Indiana Section Conference*, 2007.

[118] N. Raajan, M. Ramkumar, B. Monisha, C. Jaiseeli, and S. P. venkatesan, "Disparity estimation from stereo images," *Springer*, vol. 38, pp. 462–472, 2012. INTERNATIONAL CONFERENCE ON MODELLING OPTIMIZATION AND COMPUTING.

[119] NASA, "The computer vision laboratory." NASA Jet Propulsion Laboratory, 2003. [Online]. Available: https://www-robotics.jpl.nasa.gov/facilities/facilityImage.cfm?Facility=13-Image=335/. Last accessed, May 28th, 2019.

[120] W. Song, G. Xiong, L. Cao, and Y. Jiang, "Depth calculation and object detection using stereo vision with subpixel disparity and hog feature," in *Advances in Information Technology and Education*, pp. 489–494, Springer Berlin Heidelberg, January 2011.

[121] W. Zhao and N. Nandhakumar, "Effects of camera alignment errors on stereoscopic depth estimates," *Pattern Recognition*, vol. 29, no. 12, pp. 2115 – 2126, 1996.

[122] C. C. Yang, S. K. Huang, K. T. Shih, and H. H. Chen, "Analysis of disparity error for stereo autofocus," *IEEE Transactions on Image Processing*, vol. 27, pp. 1575–1585, April 2018.

[123] S. Poslad, *Autonomous Systems and Artificial Life*, ch. 10, pp. 317–341. John Wiley & Sons, Ltd, 2009.

[124] K. Bertels and M. R. Nami, "A survey of autonomic computing systems," in *Autonomic and Autonomous Systems, International Conference on(ICAS)*, vol. 00, p. 26, 06 2007.