Innovations in Systems and Software Engineering
https://doi.org/10.1007/s11334-020-00361-8

ORIGINAL PAPER

Autonomic architecture for fault handling in mobile robots

Martin Doran[1] · Roy Sterritt[1] · George Wilkie[1]

Received: 16 February 2019 / Accepted: 11 March 2020

Abstract
This paper describes a generic autonomic architecture for use in developing systems for managing hardware faults in mobile robots. The method by which the generic architecture was developed is also described. Using autonomic principles, we focused on how to detect faults within a mobile robot and how specialized algorithms can be deployed to compensate for the faults discovered. We design the foundation of a generic architecture using the elements found in the MAPE-K and IMD architectures. We present case studies that show three different fault scenarios that can occur within the effectors, sensors and power units of a mobile robot. For each case study, we have developed algorithms for monitoring and analyzing data stored from previous tasks completed by the robot. We use the results from the case studies to create and refine a generic autonomic architecture that can be utilized for any general mobile robot setup for fault detection and fault compensation. We then describe a further case study which exercises the generic autonomic architecture in order to demonstrate its utility. Our proposal addresses 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-automated strategy as part of its processes may result in the mobile robot continuing to function at a productive level. Our research has provided insights into the shortcomings of existing robot design which is also discussed.

Keywords Autonomic · NASA · Robots · Faults

1 Introduction

For a mobile robot to complete its tasks, it relies heavily on the performance of its hardware components. A mobile robot needs to be aware of the behavior of its components, and they are functioning within established parameters. Development of a self-diagnostic system is important, so that the mobile robot can recognize the condition of each of its components [1].

Fault detection has been in development for mobile robots since the 1970s. The field of fault detection and isolation (FDI) [2] has adapted the use of filter detectors based on Kalman filtering, to detect inaccuracies in mobile robot functions over time [3]. The use of sensor fusion [4] has also been adapted to compare expected performance models of normal sensor outputs with that of outputs from the actual mobile robot sensors. Classification and detection of faults can be established using techniques such as situation analysis [5]. Recognition of behavioral anomalies can be interpreted as symptoms of possible faults within the system. Other techniques such as redundant information statistics [6], look for subtle changes and deviations from normal execution to detect failures. Research developed by NASA in [7], explains how fault handling in (MER) Mars Exploration Rovers was implemented using (SFP) system fault protection. If faults are detected, *sequence* commands are initiated to prevent further commands being sent to the offending components. At the system level, *autonomous shutdown* commands are initiated, for example, in battery fault detection.

The focus of attention in our work is to develop a generic architectural framework for fault detection and fault compensation in mobile robots. Fault compensation is particularly important for robots operating in remote environments such as outer space, where human intervention to repair faulty hardware is not an option. Using the foundation of the autonomic computing model [8], the architectural design will concentrate on *self-monitoring* and *self-analysis* to detect

✉ Martin Doran
doran-m18@ulster.ac.uk

Roy Sterritt
r.sterritt@ulster.ac.uk

George Wilkie
fg.wilkie@ulster.ac.uk

1 Ulster University, Jordanstown, Ireland

faults and anomalies. To compensate for any faults detected, the use of *self-adjustment* will be employed. We have taken the autonomic MAPE-K feedback loop concept [8] and the three-layer IMD model [9] and expanded it to design an architecture that can handle various fault scenarios. The final generic autonomic architectural concept (Sect. 7) is a result of investigations carried out on various hardware components experiencing faults within a mobile robot. These investigations are in three areas within the mobile robot: (1) differential drive fault, (2) sensor faulting and (3) power management issues. With each investigation completed, we built up techniques for: (1) creating an intelligent monitoring process that can flag anomalies for further analysis or for reporting logs; (2) in-depth analysis processing that can make decisions on what is required to compensate for the fault found; and finally (3) we have created a policy algorithm library that can apply compensation to known faults in order to sustain a level of functionality within the mobile robot.

The paper is organized as follows: Sect. 2 describes the autonomic model in reference to the MAPE-K architecture and IMD architecture. It further examines how the MAPE-K and IMD architectures are used to develop the AIFH architecture presented in this paper. Section 3 discusses previous research that has been conducted on adapting the MAPE-K autonomic architecture and organic computing in various fields of study. This section further explores the use of *Self-Adaption* and *Fault-Tolerant Systems*. Section 4 provides some background on how hardware faults can affect the operating components within a mobile robot. It further discusses classification of faults and how these faults can impact on how the mobile robot can perform its tasks. Section 5 explores the autonomic knowledge base and its attributes. Section 6 presents the case studies that were used to create and refine the generic autonomic architecture. Various operational components within a mobile robot, such as *drive systems*, *sensors* and *power management*, are examined under fault conditions. Section 7 presents the generic autonomic architecture (AIFH), for dealing with hardware faults within a mobile robot system. It discusses the autonomic health check loop and how the System Manager and Autonomic Manager work together to allow health checks to initiate, without overwhelming the processor operations. It further discusses how the Autonomic Manager provides *Monitoring*, *Analysis* and *Adjustment* policies to deal with fault scenarios encountered by the mobile robot. Section 8 shows how the generic autonomic architecture (AIFH) can be used to deal with other hardware faults (not explored in Sect. 6), involving a stereo camera processing fault scenario as an exemplar. Sections 9 and 10 provide a summary and conclusions and discuss some future directions for this research.

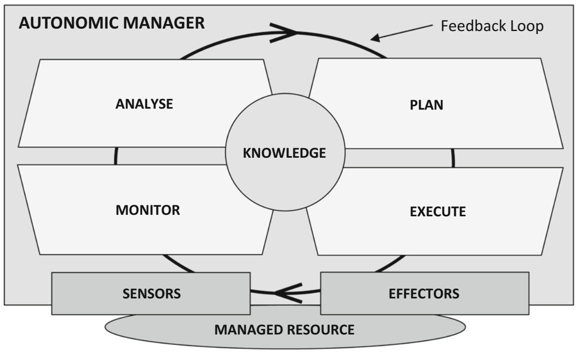

Fig. 1 MAPE-K architecture proposed by IBM [8]

2 Autonomic model and principles

2.1 MAPE-K and IMD autonomic architectures

Autonomic computing (AC) was initiated in 2001 by IBM; its aim was to develop computer systems that were capable of self-management. Computing systems over the last few years have become increasing complex, and therefore AC was proposed to take some of the decision making away from human operators and develop a self-awareness to adapt to changing conditions. The architecture associated with AC is known as MAPE (Monitor, Analyze, Plan and Execute). In self-managing autonomic systems, policies are defined to dictate the self-managing process. IBM defined four types of autonomic properties: self-configuration, self-healing, self-optimization and self-protection [8]. In research presented by [10], two modes (reactive and proactive) represent the *self-healing* process. Reactive mode is concerned with identifying a fault and, where possible, repairing the fault. The MAPE architecture was further expanded to MAPE-K: This introduced the concept of Knowledge (K) being shared between each of the four elements (Monitor, Analyze, Plan and Execute). The MAPE-K feedback loop is part of the system that allows for feedback and self-correction (Fig. 1).

IMD (intelligent machine design) is significantly different from the MAPE architecture, both structurally and behaviorally. In this alternative model, behaviors are differentiated in terms of urgency and responding to changes in the environment [9,11]. The IMD architecture closely relates to how the intelligent biological system works. The IMD architecture proposes three distinct layers: the Reaction layer, the Routine layer and the Reflection layer. The Reaction layer (lower 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 hardwired; therefore, its behavior is an autonomic response to incoming signals. The Reaction layer takes precedence over all other layers and can trigger higher layer processing. The Routine

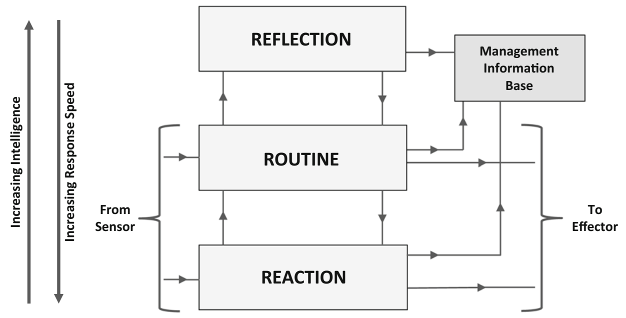

Fig. 2 Intelligent machine design architecture [9]

Table 1 Comparison of three-tier approaches

IMD	Self-managing	NASA science mission
Reflection	Autonomous	Science
Routine	Self-aware	Mission
Reaction	Autonomic	Command sequence

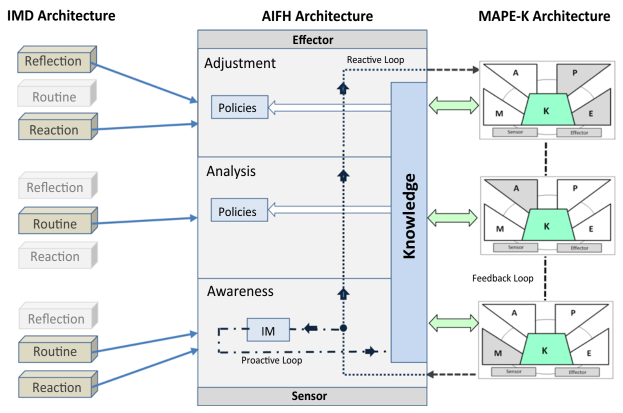

Fig. 3 Developing the AIFH architecture from the MAPE-K [8] and IMD models [11]

layer is more intelligent and skilled compared to the Reaction layer. It is expected to access working memory which contains several 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 (Fig. 2).

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 analyzes current data or historic data and identifies when to change and selects a policy to decide what to change.

In research carried out in [12], the IMD architecture describes 'Reaction' as the lowest level where no learning occurs but has direct contact with sensory systems. The middle level 'Routine' is where evaluation and planning occur. The highest level 'Reflection' is a meta-process where it deliberates about itself but has no direct contact with sensory systems. It receives data from the layer below. In Table 1, the IMD design can compare to other three-tier approaches, such as those adapted by NASA and in self-managing system. NASA places *human* labor in the top science level where the lowest level is used for command sequences to execute the mission plan (limited human intervention). Self-managing systems create a similar hierarchy, where *human* influence is stronger at the autonomous level but less so at the autonomic level [12]. The *Autonomic layer* is the bottom tier, reacts immediately to changing situations and is closest to the hardware. The *Self-aware layer* deals with daily self-managing activities, when processing capacity becomes available. The *Autonomous layer* is the top tier, system high-level objectives are directed in this layer, and it often includes *reflection* [13].

2.2 Developing the AIFH architecture based on MAPE-K and IMD principles

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*, *analyze*, *plan* and *execute* functions can be utilized [8]. The Awareness layer in AIFH requires the *monitor* function. The Analysis layer in AIFH requires the *analyze* function. The Adjustment layer requires both *plan* and *execute* functions (Fig. 3). The MAPE-K utilizes a *feedback* loop. This *feedback* loop connects the elements within the MAPE-K architecture. In the AIFH architecture, there are two 'feedback' loops. The *Reactive* loop transfers data between each layer. The *Proactive* loop is used to examine sensor data for *patterns* and *anomalies*. The *Proactive* loop provides a higher level of monitoring that is not provided in the MAPE-K architecture. The *Proactive* loop can provide status reports that can alert the user to possible impending faults within components. This could have a direct affect on a mission or task's performance, in that component faults can be identified at an early stage before they malfunction during a mission or task.

All the layers in AIFH utilize the Knowledge function which includes attributes such as policies, historical data and real-time data. The AIFH architecture also adapts a subset of 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. The AIFH architecture also incorporates the 'layer' principle, but offers all AIFH layers access to the Knowledge plane. Real-time sensor data are available at the Awareness layer. Historical sensor data are available to all layers in the AIFH architecture. The Adjustment layer is responsible for sending relevant policy data to the effectors when functionality changes are required. Compared to the IMD *reflection layer*, the AIFH *adjustment* layer can communicate with the *effector* interface directly. This allows implementation of *fault adjustment* policies to be carried out immediately, without further processing in other *layers* within the architecture.

The AIFH architecture is encapsulated within the Autonomic Manager. The principle of using autonomic management to deal with mobile robot faults allows for the possibility of identifying and evaluating the fault and, therefore through diagnosis, proposes an *adaptive* quality which allows the robot to continue to function.

3 Related work

3.1 Autonomic model: MAPE-K and organic computing

The basic autonomic architecture laid down by IBM acted as a guide to help us deal with the expanding complex systems we now find in today's technology industry. The idea was to reduce the need for human intervention and to develop a system that could make its own decisions and become self-managing [14]. Autonomic components alone are not enough; there is a requirement to integrate autonomic components with current systems. Traditionally in autonomic computing, the Autonomic Manager dictates the behavior and performance of the managed components [8]. In Organic Computing, components are not reliant on central control for coordination; therefore, the component itself can make decisions based on its observations [15]. Using an architectural model developed in [16], research carried out in [17] shows how an *hexapod* robot can detect a malfunction within its leg support mechanism. If a malfunction is detected, then the robot can initiate a leg amputation routine to discard the faulty leg. The robot will then perform a reconfiguration (*self-adjustment*), which will enable the *hexapod* robot to continue with its mission despite losing a leg.

3.2 Autonomic self-adaption: fault detection

The Autonomic Knowledge source is an area that is capable of storing configurations, policies and, most importantly, performance data past and present. This knowledge is then passed to the Autonomic Manager. Decisions made by the Autonomic Manager can be based on the type of data received from the Knowledge source [18]. Component faults do not always show themselves as simply being *non-functional* or *disabled*. In this research [19], the authors use *Evidence*, *Fault* and *Value* nodes to identify hardware faults, by recognizing changes in sensor data over time. Knowledge of how components perform over time can perform part of the autonomic knowledge base. Comparisons can be made between current component performance data and the performance data from previous tasks. Fault detection can also be achieved by comparing the performance of neighboring components of the same type. In research presented by Khalastchi et al. [20], the authors perform tests between similar components to establish if they are correlated to each other. If abnormal behavior is detected, then this could indicate a possible fault in one of the components. For this research in autonomic systems, [21] proposed that the remaining sensors could collaborate to perform a specific function if another sensor happens to fail. If a *laser range finder* sensor should fail on a remote planetary rover, that is normally used for object navigation, then engaging the *camera sensor* to detect objects could be a viable option. In this research [22], they describe how robotic failure detection, failure recovery and system reconfiguration can be achieved through their Distributed Integrated Affect Reflection Cognition (DIARC) architecture. Using an ADE multi-agent framework, they propose a system that can request information about the states of components within the network. If a failure occurs in say the navigation system, then they can locate a component to take the place of the failed component—in this case a sonar sensor array taking over from a laser range sensor.

3.3 Fault tolerance systems

When comparing autonomic methods to fault tolerance solutions, FT systems are traditionally centered around replication and exception handling. The developer needs to identify in advance the critical components and then decide what *fault* strategy to implement [23]. Research conducted by [24,25] shows the implementation of FDD (Fault Detection Diagnosis) in autonomous robotic systems. Three principles are adapted: *Timing checks*—watchdogs are incorporated to systematically check components. *Reasonableness checks*—verify test data to check the correctness of the systems variables to algorithm constraints. *Monitoring for diagnosis*—predicted behavior is computed as a specific model. This model is then compared with the observed behavior. The resulting differences, if any, are an indication of a fault. Autonomic systems are designed to look for subtle changes in behavior or inconsistent performance data. The Autonomic Element has its own Manager system. However, recent research [26] shows that systems are now being designed that incorporate both fault tolerance and autonomic principles. Research conducted in [27] shows how future NASA missions can develop *evolution* strategies to handle hardware faults. Types of failure modes include *actuator* failure, *communication* failure and *control* failure. When a fault is

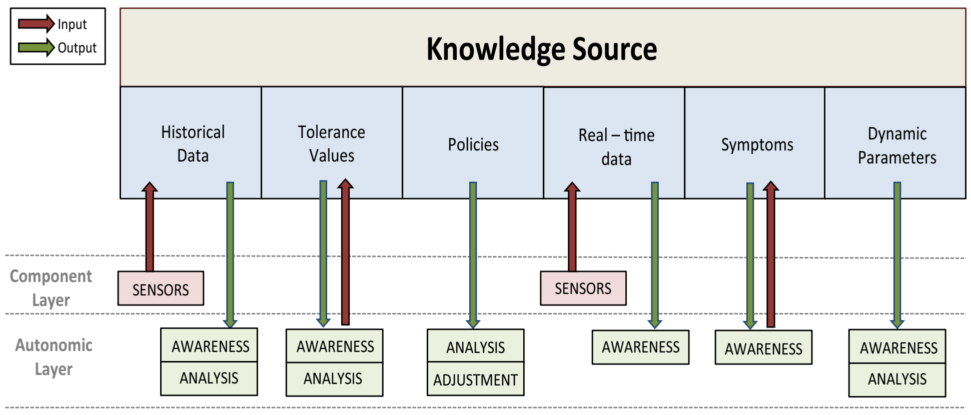

Fig. 4 Knowledge source—how *knowledge* is partitioned to reflect autonomic fault handling in a mobile robot

detected, it was proved in simulation that a rover agent adapting *Difference* evaluation methods could still maintain an acceptable performance level. In contrast, rover agents using *System* evaluation performed poorly when dealing with a fault scenario.

4 Hardware faults in mobile robots

Mobile robots are devices that rely on commands that provide instructions for 'motion' and sensor data capture that reports 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. In such circumstances, our goal is to provide fault tolerance through autonomic intervention and management. 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 some deviation from the expected behavior of the system [28]. Faults can be classified as follows: permanent (which exists until repaired), transient (which disappears on their own) and intermittent (which repeatedly appears). The severity of a fault can depend on what components in the mobile robot are malfunctioning. A major malfunction to the mobile robot drive systems would have greater impact than a major malfunction on one of the robot sonar sensors. If the mobile robot is unable to move because of a major motor failure, then its ability to carry out tasks is severely limited if not impossible; however, if a mobile robot has lost function in a single sonar sensor, it may still function even with reduced sensor ability.

In the real world, 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 [29]. Each of the six rubber wheel casings on the Rover had been punctured by sharp rock material from the planet's 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. In 2006, NASA reported that the Spirit Rover had suffered a broken wheel and was beyond repair due to circuit failure [30]. Any further trips conducted by Spirit meant that Mission Control had to map out a route that avoided terrain with loose soil.

5 Knowledge source

In IBM's Autonomic Blueprint [18], the 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, a *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 by [31], the authors use the *Knowledge Base* to store *Recovery Patterns*. When a component failure occurs, the Autonomic Manager will then select the appropriate recovery pattern(s) to compensate for the fault.

Figure 4 shows how the *knowledge source* can be implemented in autonomic fault handling for a mobile robot. Sensors provide input data to the *knowledge source*. The output data are used by the Autonomic Manager (within the AIFH architecture) and distributed to the Awareness, Analysis and Adjustment layers when required. As the mobile robot performs its tasks, all sensor data are recorded so that historical behavior patterns can be analyzed. Tolerance values can be stored here, so that faults can be identified if tolerance limits are exceeded. Policies used to analyze fault data and adjust for faults can be stored in the knowledge base. As the robot performs its tasks in real-time, 'live' data can be recorded here and compared with historical data. If sensor readings are trending toward 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. Finally, 'dynamic parameters' can be stored in the *knowledge source* and are available to the Awareness and Analysis layers.

6 Case studies

The following case studies are presented in the order they were conducted and used to influence the evolving AIFH architecture. Using software engineering techniques, these case studies contribute to engineering of self-managing systems, where there has been relatively little research, as identified by Sterritt [32].

6.1 Study one: robot wheel alignment fault

6.1.1 Error detection

This case study centered on investigating the failure in the *effector* systems of a mobile robot, centering on wheel alignment accuracy [33]. The robot must become *aware* that there is a problem with its differential drive system. When the robot arrives at its destination, there is a *health check* procedure to determine if it has arrived at the expected destination. If the *health check* procedure reports that the robot is not at the expected destination point, then further procedures would be put in place to analyze and determine the extent of the fault.

Experiments were conducted using a Pioneer P3-DX mobile robot fitted with an LMS 200 Laser [34]. The first part of the experiment was performed using wheels that were in perfect working order. The mobile robot was instructed to move a fixed distance up and down the laboratory in a parallel path to a wall. At the start of each run, the mobile robot would record the laser distance reading (from robot to wall) and would repeat this when the robot came to the end of its journey. This was repeated multiple times to give an average wheel alignment performance reading. This information or data are stored for reference later. For the second part of the experiment, the robot was fitted with a damaged wheel. The robot was then put through the same testing as in the first experiment. As the robot performed the tasks, self-monitoring was initiated to evaluate the data from the robot. Figure 5 shows how a wheel fault has affected the robot's alignment tracking when attempting to drive in a straight line. The consequences of this fault are that the robot will not arrive at its expected destination point and this will inhibit any tasks assigned to the robot. We describe the wheel alignment fault in three stages: (1) Awareness—the discovery of the wheel alignment fault by using *dead reckoning*, (2) Analysis—allows us to establish the extent of the fault, and (3) Adjustment—establishes a method that could be used to compensate for the fault.

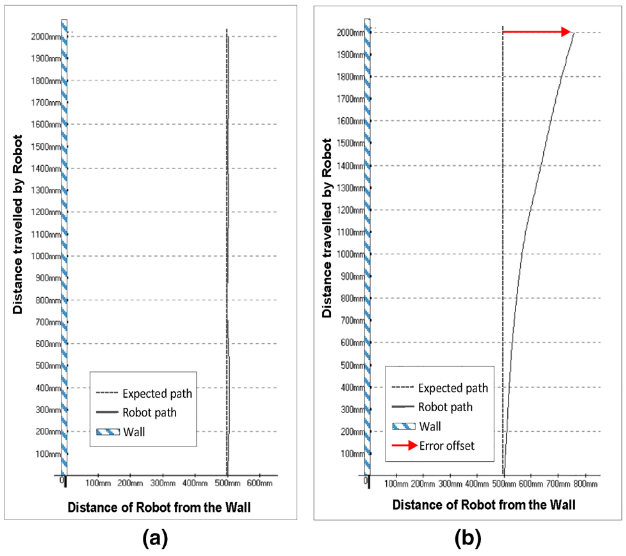

Fig. 5 Graph (**a**) shows the path of the robot with both wheels at optimal performance. Graph (**b**) shows the path of the robot with one wheel in a damaged state

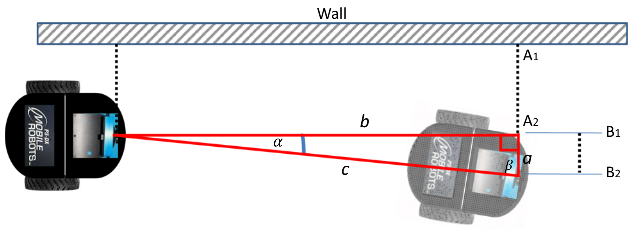

Fig. 6 The Pioneer P3-DX robot with a damaged wheel—calculating the value that represents the error angle α

6.1.2 Error evaluation

Now that the robot is slewing away from its expected destination point, we had to find a method of turning the robot back toward its expected route path. Using the data gathered from the experiments represented in Fig. 5b, we calculated the distance the robot traveled and calculated the distance the robot was from its expected destination point. Figure 6 shows how we established the *angle error* value using trigonometry.

6.1.3 Error adjustment

We have established the *error angle* value α and can use this value to calculate the *angle of turn* needed to rotate the robot back toward its expected path while traveling to its destination point. The *angle of turn* calculation is established

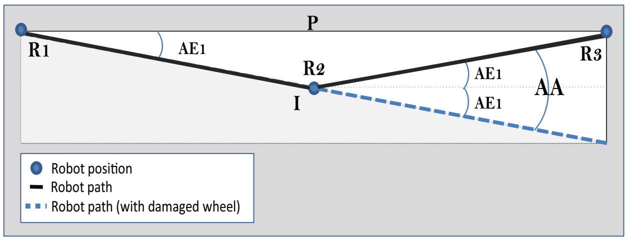

Fig. 7 Representation showing how the angle of turn is calculated to realign the robot path

using the values represented in Fig. 7. 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 AE1 represents the angle of the wheel alignment error calculated in Fig. 6. The AE angle value is then doubled. The reasoning behind this is that twice the AE1 values is required to bring the robot back to the expected path. The 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. [Equation (1)]. The robot continues its journey by moving forward on its new heading for another interval.

$$\mathrm{AA} = \frac{2\mathrm{AE}}{I} \tag{1}$$

The more the *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, but this is traded off against speed. In Eq. (2), the *interval distance* is represented by ID and total distance is represented by TD. The interval distance is calculated as follows:

$$\mathrm{ID} = \frac{\mathrm{TD}}{I} \tag{2}$$

The compensation method described in Fig. 7 reflects the ability of the robot to *self-adjust*. This autonomic *self-adjustment* allows the robot to arrive close to the expected destination point, even with a damaged wheel. With the calculations for the *wheel alignment error* established, an algorithm was developed (Algorithm 1) and tested with the Pioneer P3-DX robot.

As a result of implementing Algorithm 1, Fig. 8 shows how the compensation method improves the robot's ability to track close to its intended path. The greater the number of *intervals* employed results in a decrease in the *error offset* value. The severity of the wheel alignment error will influence how the robot will perform over long distances. If the wheel alignment error is considerable, then the number of *intervals* required for the robot to stop and adjust itself will increase; this could have an impact on resources such as power consumption and task time.

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
Output: The angle of adjustment required
        ($AngleOfAdjustment$) to turn the robot when an
        interval.
$cd$ = current distance traveled by robot
$iv = dis/ni$
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 (iv);
    if ($dis$ mod $iv = 0$) then
        StopRobot();
        RotateRobot($aa$);
        MoveRobot();
    end
    $cd$ = updateCurrentDistanceTravelled();
end
```

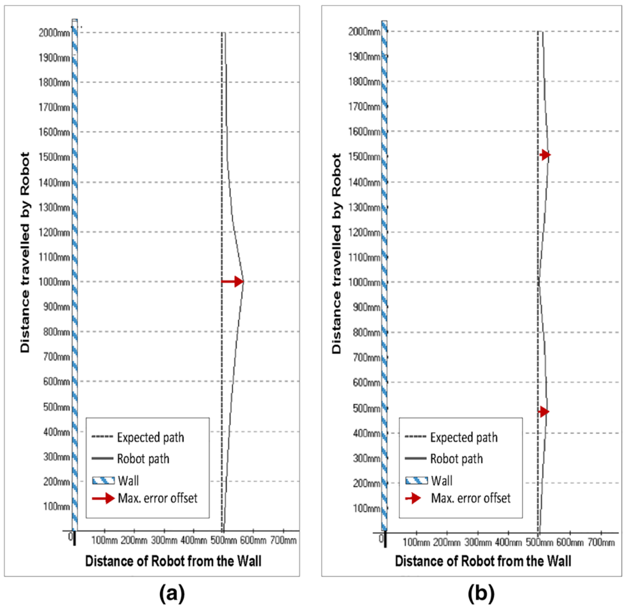

Fig. 8 Using the compensation algorithm, the robot journey accuracy is increased when the number of intervals is also increased. **a** Robot journey using one interval. **b** Robot journey using two intervals

6.1.4 Developing the architecture: Part 1

The autonomic architecture for the fault handling model is constructed using three layers: *Awareness*, *Analysis* and *Adjustment*. The Autonomic Manager manages the communication between each layer and how the *knowledge base* is shared. In the MAPE-K architecture [18], the Autonomic Manager implements an intelligent control loop that is made up of four parts. Each part communicates and collaborates with one another and shares 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 passes through each layer within the fault handling architecture. This control loop is responsible for passing fault data between each layer.
- *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.

In this case study, the basic autonomic architecture for the wheel alignment fault is presented. Figure 9 shows the Autonomic Manager which contains the three-layer AIFH functionality. Within each functional layer, the Reactive Control loop controls the flow of the wheel alignment fault data. The fault data are first collected within the *Awareness layer*. In this layer, a decision is made to whether a fault has occurred. The Proactive Control loop will check for unusual readings from historical and current data that are provided by the *knowledge base*. The fault data are then passed to the Analysis layer to calculate the extent of the fault. The Analysis layer uses the *knowledge base* to select the relevant policy to analyze the fault data. Fault calculations provided by the Analysis layer are then passed to the Adjustment layer. The Adjustment layer will select a relevant policy with the *knowledge base* and use the fault calculations from the Analysis layer to perform the necessary fault adjustment.

6.1.5 Summary

In this case study, we discovered that awareness/analysis of past performances enabled us to establish the extent of the wheel alignment fault. A basic autonomic architectural model is introduced to handle the wheel alignment fault. The importance of *knowledge* is key when determining (1) if there is a fault (awareness), (2) the extent of the fault (analysis) and (3) what is required to compensate for the fault (adjustment).

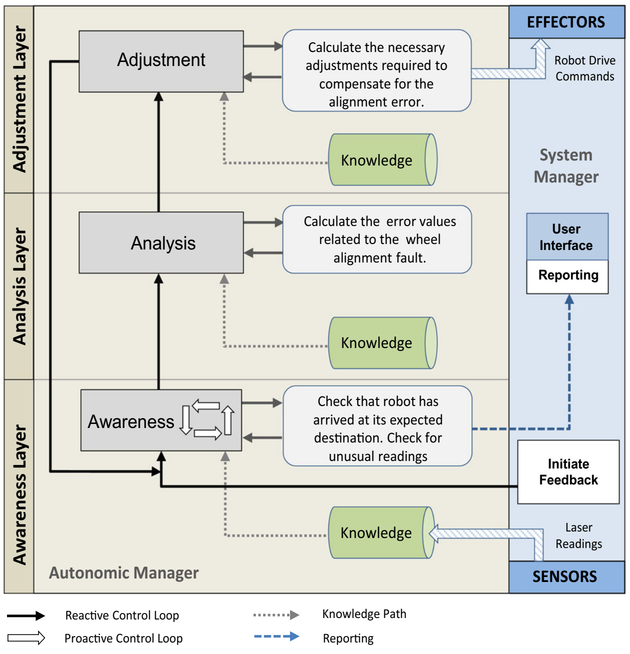

Fig. 9 Basic autonomic model showing how the *wheel alignment fault* is handled by the AIFH architecture

6.2 Study two: robot sonar sensor faults

In this case study, we examined the effects of losing one or more sonar sensors in a mobile robot [35]. The experiments were carried out using a Pioneer P3-DX robot fitted with a sonar sensor array (Fig. 10). The autonomic *self-adaptive* approach to handling fault scenarios assumes that even with reduced sensor capability, it is still possible to carry out mission objectives. Faults in sonar sensors can manifest in two ways: (1) The sonar sensor stops reporting data (the sensor has been damaged or there is an electronic failure). (2) The sonar sensor is reporting data, but these *data* are unreliable (due to minor malfunction or minor physical damage). When a sonar sensor becomes faulty, then it will affect the ability of the robot to detect objects in its path. The loss of one sonar sensor has limited impact, but the loss of several sensors will severely reduce the robot's ability to detect objects.

As shown in the previous case study (Study one), we consider the fault process as *Awareness*, *Analysis* and *Adjustment*. Through monitoring and knowledge gained from previous tasks, the robot can become *aware* that there is a possible fault with the sonar sensors. In the *Awareness* process, faults can be detected through unusual sensor readings or by a *reactive* process where the sensor sends out a 'dead' signal. If a fault is flagged, then the robot system processing can do in-depth analysis to establish the extent of the

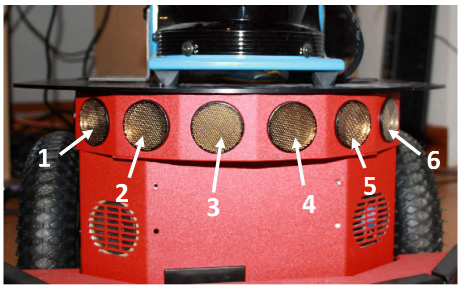

Fig. 10 The sonar sensors on the Pioneer P3-DX robot are arranged as an array 1–6

fault. It can then use data gathered from the analysis procedure to determine what possible *Adjustment* can be made to compensate for the fault.

For this experiment, we were only concerned with sensors (1–6); these are the 'forward' facing sensors. See Fig. 10. The failure *states* for the sonar sensors on the Pioneer P3-DX are classified as follows:

Sonar Sensor Failure States:

- IsNormal—all sonar sensors are working as expected.
- IsMinor—one or two sonar sensors are either disabled or reporting erroneous data
- IsMajor—a loss of three or more (but not all) sonar sensors. Provides only limited sensing ability.
- IsCatatrophic—all forward-facing sonar sensors are disabled. No ability to detect objects.

6.2.1 IsNormal state

In Fig. 11a, we tested all the sonar sensors under normal conditions. This *IsNormal* state proved that each of the sensors was able to detect an object correctly. An object was placed in front of the P3-DX robot. Measurements were taken between the object and each sonar sensor using measuring tape. These values were then compared to the values being reported by the sonar sensors to establish if the sensors were operating as expected.

6.2.2 IsMinor state

In Fig. 11b, if a sonar sensor has become faulty (due to impact or electrical fault), then it signals a default reading to the System Manager program as '5000.' The Autonomic Manager (Awareness) process uses the *knowledge base* to establish that this is a sensor fault. The sensor fault data are then passed to the Analysis layer for processing. Faulty sonar sensors can also be detected using a *proactive* feedback loop. In this experiment, we compare the values reported by neighboring sonar sensors. If a sonar sensor is reporting significantly different data (within a tolerance range) to its neighboring

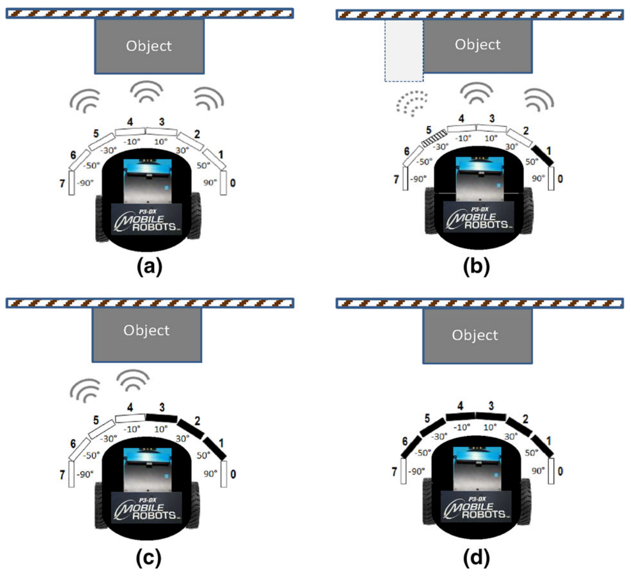

Fig. 11 Failure states for the sonar sensors on the P3-DX mobile robot

sensors, then we can establish that this sensor's data cannot be relied upon; this sensor is then marked as being 'disabled.' Algorithm 2 shows how readings from neighboring sonar sensors are used to test if a sensor is reporting correct object detection data.

6.2.3 IsMajor state

In Fig. 11c, when two or more sonar sensors become faulty, 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 of the sonar sensors in the 'array.' 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 this occurs, the autonomic analysis procedure is employed to identify what sonar sensors are faulty.

6.2.4 IsCatastrophic state

Figure 11d shows that this state reports that all sonar sensors are disabled. When all sonar sensors are reported as disabled, the robot is automatically stopped; this is to prevent any unnecessary damage to the body of the robot.

6.2.5 Sonar sensor fault compensation

Within the *Adjustment* layer, a Compensation Policy is initiated to deal with any failure found in the six forward-facing sonar sensors. To compensate for a faulty sonar sensor, we employ a 'stop' and 'rotate' strategy. The remaining fully

Algorithm 2: Highlight Disparate Readings Between Adjacent Sonar Sensors

```
Input: sonarReadings sr[] = readings from 1–6 sonar sensors
       toleranceRange tr = tolerance value allowed between
       adjacent sensors
       sonarPosition 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
```

functional sonar sensors are used as substitutes for any faulty sensor. The more the sonar sensors that are faulty, the more 'stop' and 'rotation' commands will be required to detect objects. Using the six sonar sensors in the array, there are sixty-four possible combinations using binary notation. Combination 1 = 000000 (this indicates all sonar sensors are working correctly and no action is needed). Combination 64 = 111111 (this indicates all sonar sensors are disabled); no compensation can be deployed when the robot is in this state. This leaves sixty-two fault combinations that can be compensated for. The mobile robot will need to rotate (clockwise or anticlockwise), in order to compensate for loss of some of the sonar sensors. The position of each sonar sensors on the array is indicated as 1–6 (Fig. 12). The angle between each of the sonar sensors is 20°; therefore, all rotations are done in single or in multiples of 20° values (Fig. 12). A single sonar sensor fault will only require one rotation of the mobile robot.

If there are multiple sonar sensor faults, then the number of rotations will increase. Table 2 shows the sonar fault scenarios 'tests' and 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 for Disabled Sonar Sensors* algorithm is presented in Algo-

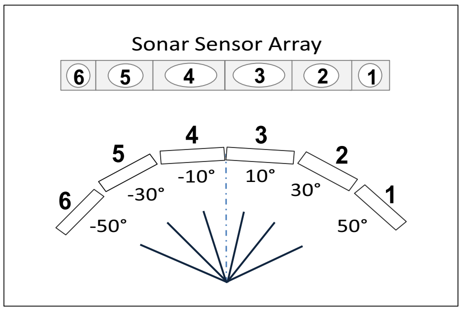

Fig. 12 The sonar sensors are arranged 1–6 on the array with a 20° angle between them

Algorithm 3: 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
```

Table 2 Sonar sensor fault scenarios

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

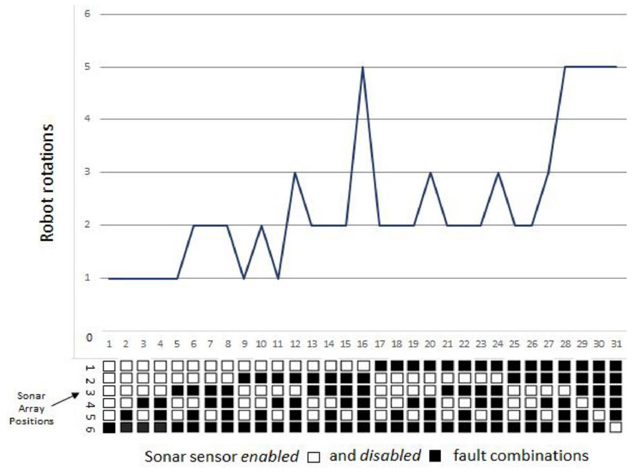

Fig. 13 The increase in the number of sonar sensor faults will also increase the number of rotations required to compensate for the fault

rithm 3 and Algorithm 4. Algorithm 3 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 4 is used to work out the minimum number of robot *rotations* required to compensate for the faulty sonar sensors. Finally, all rotation values are stored in a program *array*, so that they can be fed to the robot’s System Manager (via the Compensation Policy), to execute the physical rotations required to compensate for the faulty sonar sensors.

In Fig. 13, a selection of possible fault combinations is displayed (showing and average 31 *alternate* combinations of the possible 62 combinations). The greater the number of ‘disabled’ sonar sensors on the array, the greater the number of robot rotations required to compensate for the faulty sensors. This ultimately will have an impact on task time and power required.

Algorithm 4: 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 (disabledArray.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
```

6.2.6 Refining the AIFH architecture

In the previous case study (wheel alignment fault), the basic autonomic architecture for fault handling was presented. Using this case study (sonar sensor fault management), we

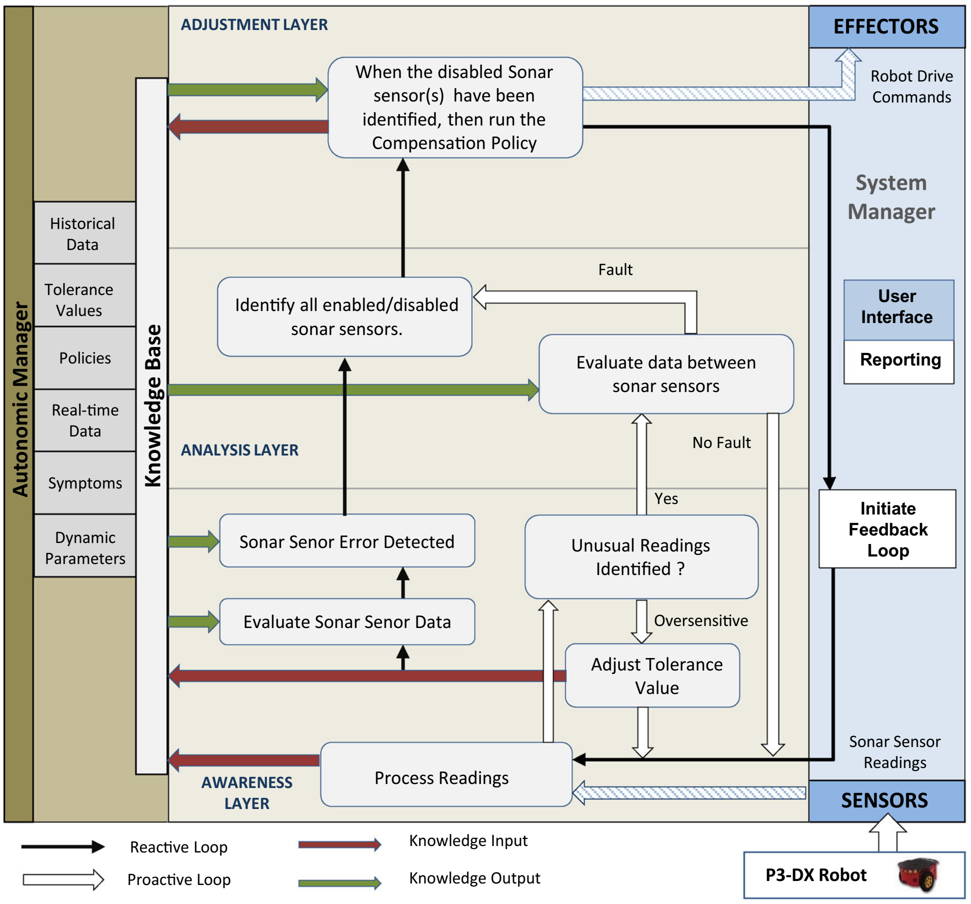

Fig. 14 Awareness and Analysis layers—part of the AIFH architecture, showing how the sonar sensor faults are handled

will focus on the *Awareness* and *Analysis* layers of the AIFH architecture (Fig. 14).

Task data from the sonar sensors are processed and updated to the knowledge base. As each task is performed by the mobile robot (P3-DX), the sensor data are recorded. These records will then collate to form the *historical data* within the knowledge base. The Reactive Control loop evaluates the current sonar data and uses the *tolerance* values stored in the knowledge base, to establish if any of the sensors are showing any unusual behavior. The knowledge base *real-time* sonar data are compared to the stored *Dynamic Parameters*, which will, in turn, identify if any sonar sensors are *disabled*.

The Reactive Control loop then passes the sensor fault data to the *Analysis* layer for processing. The Proactive Control loop processes the sonar data and establishes if any of the sonar sensors are reporting unusual readings. Sonar sensors are flagged if the *distance-to-object* readings between neighboring sensors are greater than the expected tolerance values. The Proactive Control loop will then pass those sonar sensors marked as 'unverified' to the *Analysis* layer for further investigation. If the *Analysis* layer identifies a fault within a sonar sensor, the knowledge base is updated and the sensor is marked as *disabled*.

6.2.7 Summary

By introducing a *compensation* policy to handle sonar sensors faults, the Pioneer P3-DX robot can still detect objects. However, the more the sonar sensors that are at fault, the greater impact it will have on the operational efficiency of the robot.

When a fault occurs in the sonar sensor array, then a *sonar failure* mode is engaged. Figure 15 shows how the robot is instructed to stop at every 200-mm 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. In this case study, we also investigated the role of the *Awareness* layer within the Autonomic Manager (Fig. 14). The Reactive and Proactive Control loops play key roles in establishing sonar sensor faults by using the shared *knowledge base* data. The Proactive Control loop can make adjustments to tolerance levels during the *Analysis* process, depending on the position of the sonar array to the *object* being detected.

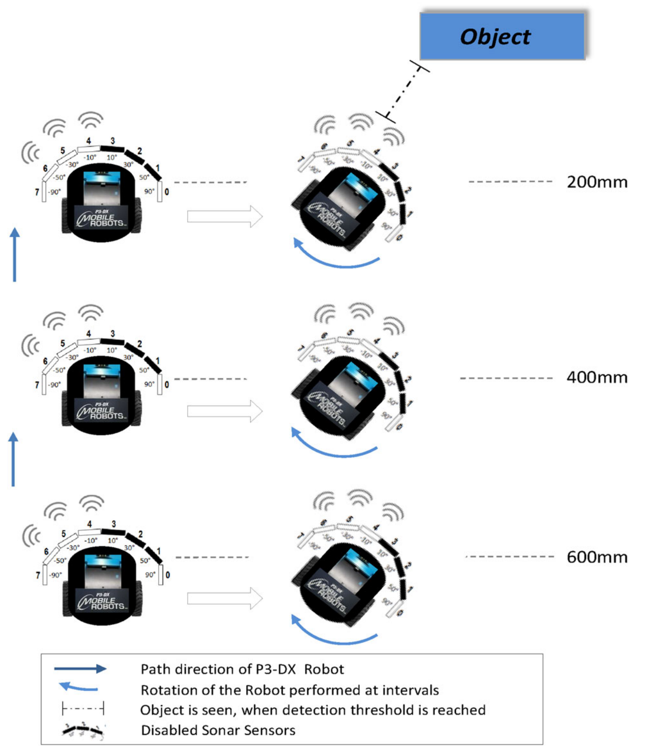
Object
200mm
400mm
600mm
Path direction of P3-DX Robot
Rotation of the Robot performed at intervals
Object is seen, when detection threshold is reached
Disabled Sonar Sensors

Fig. 15 When a sonar fault is detected, the robot is stopped at selected intervals. The robot is rotated to check for possible objects

6.3 Study three: robot battery degradation fault

In this case study, we examined the effects lead–acid battery degradation [36] has on the power resource management on a simulated Pioneer P3-DX robot [37]. Although this type of failure is evident in all lead–acid batteries, we decided to use a simulated battery setup for this experiment as battery degradation is very difficult to recreate in laboratory conditions using a real battery.

6.3.1 What is battery degradation?

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. 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 [36]. DOD (Depth of Discharge) is used to describe how deeply a battery is discharged. The less a battery is discharged, the greater the number of 'charge' cycles you will get from the battery over its lifetime. Figure 16 shows the DOD characteristics of the lead–acid battery used in the Pioneer P3-DX [38].

The Pioneer P3-DX robot contains components that require a certain level of power input. The lead–acid batteries

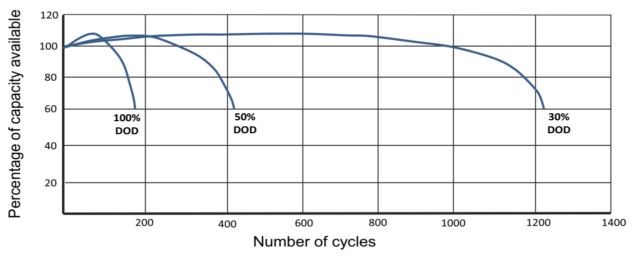

Fig. 16 The DOD (Depth of Discharge) characteristics for the lead–acid battery used in the Pioneer P3-DX [38]

Table 3 Power requirements for each component in the Pioneer P3-DX robot

Component	Power (W)	Percentage (%)
Motion	2.8–10.6	12–44.6
Sensing (sonar)	0.58–0.82	1.9–5.1
Microcontroller	4.6	14.8–28.8
Embedded computer	8–15	33.3–65.3

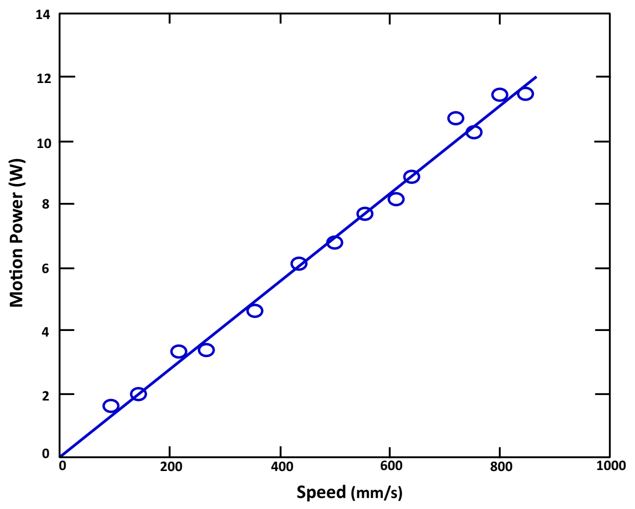

Fig. 17 Research in [39] shows the power (W) required for the 'Motion' component in the P3-DX when driven at various speeds

contained within the robot supply the necessary power for the components. Research conducted in [39] shows the relative power required for each of the Pioneer P3-DX components (Table 3).

6.3.2 Experiments: battery degradation effects

For this experiment, we investigated how battery degradation can affect how much power is available for the 'Motion' component during different stages of the lifetime of the battery. Figure 17 shows how the research conducted in [39] describes the amount of power needed for the robots 'Motion' component when driven at various speeds.

Table 4 Robot task: setup values for robot running @ battery cycle 0

Parameters	Values
RS Robot speed (mm/s)	600
PR Power required (W)	8
DIS Distance to travel (M)	5000
T Time (h)	2.31
WHU Watt-hours used	18.48
WC Watt-hour capacity	25.2
BC Battery cycle	0

Table 5 Robot task: setup values for robot running @ battery cycle 1100

Parameters	Values
RS Robot speed (mm/s)	800
PR Power required (W)	8
DIS Distance to travel (M)	5000
T Time (h)	2.31
WHU Watt-hours used	18.48
WC Watt-hour capacity	22.68
BC Battery cycle	1100

The architectural design for the power management of the robot involved a System Manager and a Autonomic Manager [37]. 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 analyzes 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.

6.3.3 Initial task setup

Using data collected by research conducted in [36], the battery data from Fig. 16 and the user input values, we can construct parameter and test values (Table 4).

The battery in the Pioneer P3-DX provides 84 watt-hours of power capacity [40]. When the battery is at cycle 0, the battery capacity is 100%. The battery rating is 7000 mAh offering 12 volts [38]. We can therefore use Eq. (3), 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 [40].

$$E(wh) = Q(mAh)\ x\ \frac{V(v)}{1000}. \tag{3}$$

Using the values in Table 4, we can establish the value of the 'Motion' component in terms of *watt-hours* used.

$$WHU = PR\ x\ T. \tag{4}$$

To prolong the life of the battery, we employed a DOD rate of 30% (Fig. 16). 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 we use 30% DOD rate, then we have 25.2 watt-hours available for the Pioneer robot at battery cycle 0—see Eq. (5) for how watt-hour capacity **WC** is calculated at the DOD rate. Using the **WHU** value from Eq. (4), we can then calculate the percentage capacity (**PC**) required for the robot to complete robot task (Table 4).

$$WC = \frac{E(wh)}{100}\ x\ DOD \tag{5}$$

$$PC = \frac{WHU}{WC}\ x\ 100. \tag{6}$$

The **PC** value is calculated using Eq. (5). For this experiment, the acceptable *threshold* value for how much capacity a robot task uses is set to 80% (**AT**). If the **PC** value is below the **AT** threshold 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 robot task (Table 4) requires a battery charge capacity **PC** of 73.33%, when employing Eq. (6). The **PC** value of 73.33% is below the threshold value (80%), and therefore no adjustment from the Autonomic Manager is required.

6.3.4 Executing a robot task with battery degradation

Toward the end of a battery's lifetime, the amount of capacity is reduced. The next evaluation shows how a robot task performed using the same values as in Table 4 will be affected by using a battery in a later 'cycle' state. The robot task is run using cycle 1100 (Table 5), which results in capacity dropping from 100 to 90% (Fig. 16). We need to re-calculate the (Ewh) value using Eq. (3). 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 Eq. (6), then the task will require 81.48% of battery capacity **PC**, which is above the acceptable threshold **AT** value of 80%.

Algorithm 5: Check that the battery 'Percentage Capacity' 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
```

6.3.5 Applying compensation during battery degradation

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, the **WHU** value is now at 18.00 using Eq. (4). We can then calculate the **PC** value using Eq. (6). 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.

Algorithm 5 shows how task input and analysis performed by the Autonomic Manager can establish if the **PC** value is within tolerance values. If the **PC** value is above the acceptable threshold limit, then Algorithm 6 is initiated. The Compensation Algorithm 6 will be periodically run until the **PC** value is above the threshold limit value.

Algorithm 6: 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
```

Table 6 Robot task: compensation—reduce speed @ battery cycle 1100

Parameters	Values
RS Robot speed (mm/s)	500
PR Power required (W)	6.5
DIS Distance to travel (M)	5000
T Time (h)	2.77
WHU Watt-hours used	18.00
WC Watt-hour capacity	22.68
BC Battery cycle	1100

6.3.6 Refining the AIFH architecture

In the previous two case studies, we introduced the outline design for the generic autonomic architecture. Case Study One explained how the three layers of the AIFH architecture would be integrated with the knowledge base. Case Study Two showed how the autonomic *feedback* loop was implemented within the *Awareness* and *Analysis* layers and its interaction with *knowledge base* inputs and outputs. In this architectural refinement section, we focus on the *Analysis* layer and the *Adjustment* layer. In this case study scenario, the Awareness layer is constantly checking the 'cycle' count and the DOD value. The *knowledge base* 'dynamic parameters' are constantly updated to reflect the 'cycle' count of the battery. Tolerance values stored in the *knowledge base* allow the *Awareness* layer to know when battery degradation has begun. When this occurs, the Reactive Control loop will pass data to the Analysis layer to check if the robot task can be completed using the current battery capacity. Figure 18 shows how the *Analysis* layer uses the *knowledge base* policies in order to test if the battery capacity is within expected tolerance ranges. The tolerance values are also stored within the *knowledge base*. If tolerances are not exceeded, then the Reactive Control loop will report back to the Awareness layer that the battery status is stable. However, if tolerances are exceeded, then the Reactive Control loop passes data to the *Adjustment* layer. Within the *Adjustment* layer, policies

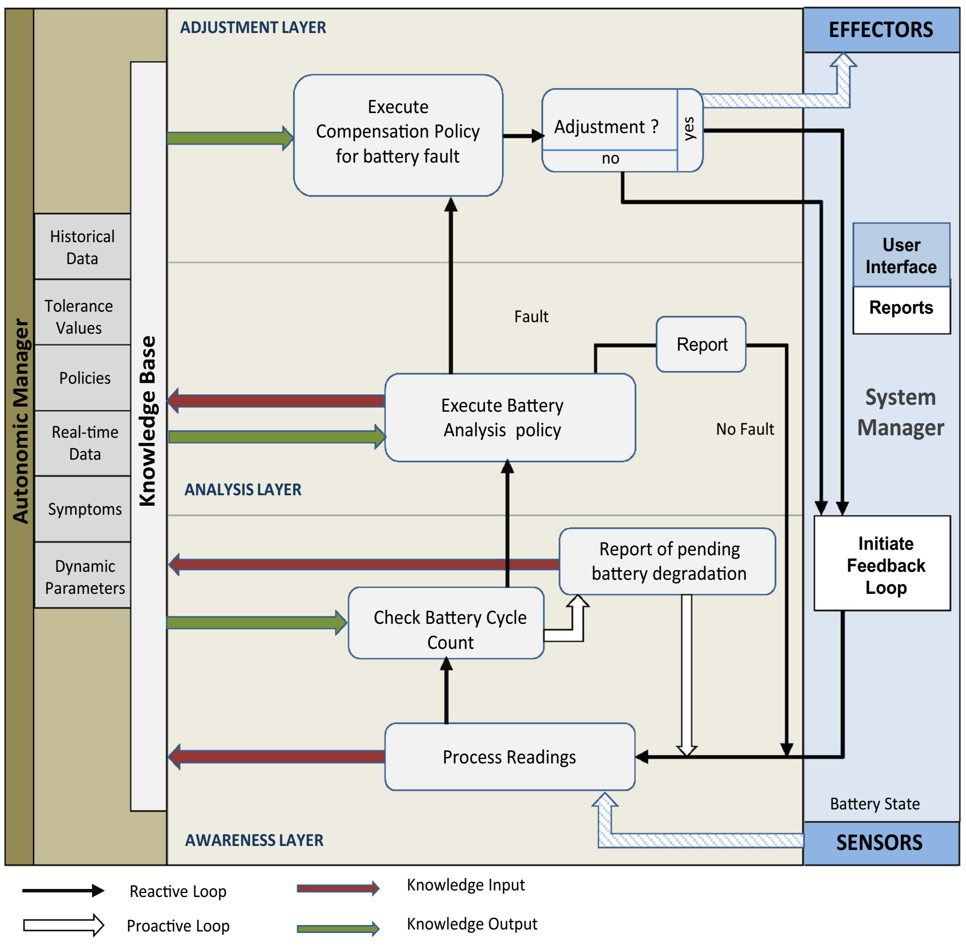

Fig. 18 Architectural development for battery degradation fault. How knowledge base is implemented in the analysis and adjustment layers

stored in the *knowledge base* are implemented to adjust the speed of the robot and therefore reduce battery consumption. The Reactive Control loop will then return control to the System Manager, where it will re-engage the *Awareness* layer, so that the cycle can begin again when the Autonomic Manager re-initiates a fault health check.

In this version of the refined AIFH architecture, the Autonomic Manager makes use of the *dynamic parameters* stored in the knowledge base. The System Manager updates the battery 'cycle' count to the *dynamic parameters* store within the knowledge base. The Analysis layer then can use the tolerance value check *process* although with the battery 'cycle' count to establish if the battery has entered into its *degradation* phase.

6.3.7 Summary

In this case study, we examined 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, where 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 '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. 16) and checking the *cycle count* of the lead–acid battery in the robot. The 'Analysis' process can cross reference the cycle count with the percentage of charge available in the battery. The data from the 'Analysis' process are then made available to the 'Adjustment' process. The 'Adjustment' policies can then calculate what power reduction in certain components is required, therefore reducing battery consumption.

7 Generic autonomic architecture for fault detection (AIFH)

With the case study investigations completed in Sect. 6, we were able to combine all the knowledge and lessons learned in the tests and development, to design a generic autonomic architecture for fault management. The aim of the generic autonomic architectural design is to handle various types of component faults which include *fault detection*, *fault analysis* and *fault recovery*. 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 System Manager controls the flow of data from the robot's sensors and effectors. The System Manager also handles tasks performed by the robot and initiate the *heath check* autonomic control loops. The overall architecture is presented later in Fig. 21. The following text is used to explain the roles and responsibilities of a number of components prior to seeing them within the overall architectural diagram (Fig. 21).

7.1 System Manager

The System Manager is responsible for controlling the sensors and effectors of the mobile robot. A *Task Module* within the System Manager is used to contain the task data and task commands. The *Task Module* initiates the sensors and sends command operations to the effectors. Data from the sensors are then processed using the *Sensor Processing Module*. The *Sensor Processing Module* is then responsible for updating the *knowledge base* with all the sensor's readings accumulated during the task operation. As each task is performed, the System Manager initiates the *Autonomic Control Loop* using the *Health Check Module*. This operation is performed at intervals during the task or mission. The Autonomic Manager takes control of the processing to check that all sensors and effectors are performing within tolerance limits. The System Manager also contains an *Output Module* which is used to relay data to Users or Mission Control, regarding fault diagnosis, symptoms of possible impending faults and fault recovery information.

7.2 Autonomic Manager

The Autonomic Manager is focused on the administration of software systems and therefore handles the complex tasks that would normally be handled by the System Manager. When the System Manager initiates the *Health Check Module*, the Autonomic Manager takes control of the task processing. In the AIFH architecture, we employ two control loops—Reactive and Proactive. Research developed in [41] 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 three layers (Awareness, Analysis and Adjustment). The Proactive Control loop operates within the Awareness and Analysis layers. The *knowledge base* module is available to all three layers within the Autonomic Manager.

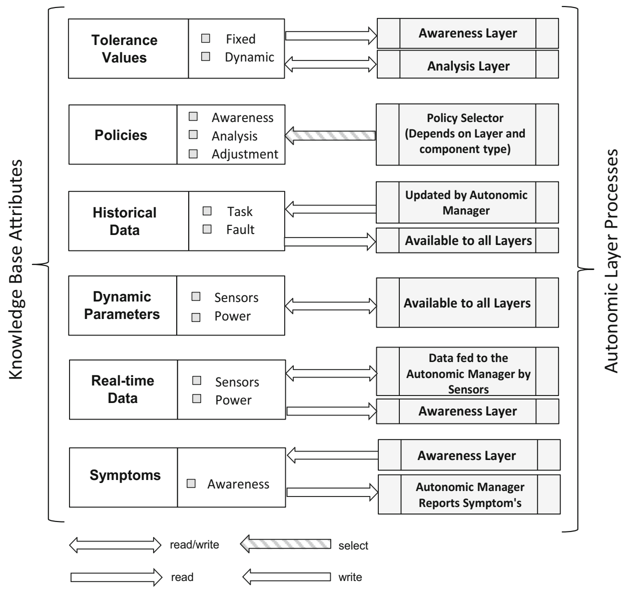

Fig. 19 Representation showing how the attributes within the knowledge base are used by each layer within the AIFH architecture

7.2.1 Knowledge Base Module

The *Knowledge Base Module* provides each layer in the AIFH architecture with historical task data, tolerance values, policies, real-time data, symptoms and dynamic parameters. The *Knowledge Base Module* is dynamic and is constantly updated with sensor data supplied by the executing task. Figure 19 shows how each attribute within the *Knowledge Base Module* is 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 *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 is used by all three layers in the AIFH architecture. Some policies are used to check if sensors are operating within tolerance limits. Other policies involve analyzing data to establish the extent of a fault. Policies are also available that can adjust the behavior of the robot to compensate for a fault. Historical data are constantly updated by the Autonomic and System Managers. Historical data are important in order to track behavioral changes within the robot's components. 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.

7.2.2 Building the AIFH architecture

In Sect. 6, the case studies provided methods for detecting and analyzing faults. There were also methods to adjust for those faults. These experiments provided a foundation to develop the AIFH architecture. Figure 21 shows the integration of the System Manager, Autonomic Manager and knowledge base, as a fully formed generic autonomic architecture. The System Manager controls the timing of the *Health Check* monitoring, and the Autonomic Manager orchestrates how the *feedback* loops traverse each layer *Awareness, Analysis and Adjustment*. The knowledge base is shared by each of the AIFH layers to provide *sensor*, *historical*, *tolerance* and *parameter* data in order to detect/adjust for component faults.

7.2.3 Awareness layer

As the mobile robot executes its allotted task, 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 be used for the current robot task. 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* later 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 these 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 User Interface or Mission Control via the System Manager. These reports are vital and could prevent future tasks being compromised.

7.2.4 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*. The analysis policy 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 labeled 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 m, 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 m.

If *tolerance* adjustment is required, the Reactive Control loop will redirect back to the *Awareness layer* for re-evaluation. Once the fault calculations are made in the *Analysis layer*, then the fault parameter data are passed to the *Adjustment layer*.

7.2.5 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. 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, an adjustment calculation can be made, i.e., if there is at least one sonar sensor still operable, then a *adjustment* policy can be deployed.

When the *adjustment* policy is deployed, the Reactive Control loop will send the *compensation* parameters to the System Manager (Task Module). The *Task Module* will then direct the effectors/sensors to operate using the new *adjustment* policy. To test the *compensation* strategy is successful, the *Health Check Module* will then re-initiate the control

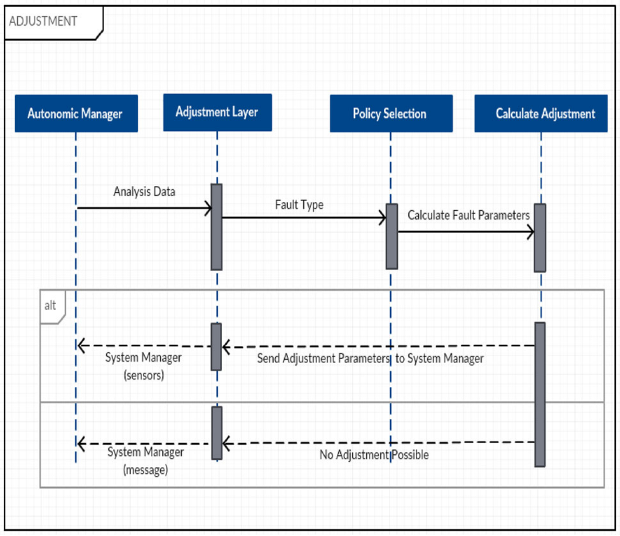

Fig. 20 UML sequence diagram showing the relationships within the AIFH adjustment layer

loops in the Autonomic Manager (Awareness layer). The *Component Health Check* in the *Awareness layer* will check the component against current tolerance limits. If the *adjustment* policy is successful, the Reactive Control loop will redirect back to the *Health Check Module* in the System Manager to report that no faults are pending. Figure 20 shows an UML Sequence representation of the modules and process routes within the Adjustment layer.

7.2.6 AIFH autonomic architecture summary

The AIFH architecture was initially developed using principles found in the MAPE-K and IMD architectural models [8] [9]. Further development of the AIFH architecture was achieved by using the research carried out in the case studies performed in Sect. 6. Figure 21 shows how the Autonomic Manager (containing the three layers, *awareness*, *analysis* and *adjustment*) is integrated with the System Manager. The System Manager is responsible for executing commands to the mobile robot via the *effectors* and *sensors*. The System Manager is also responsible for running *task* procedures and *health check* monitoring. The Autonomic Manager provides a mechanism for detecting faults, analyzing faults and providing policies that can compensate for faults. 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 certain component behaviors 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 continu-

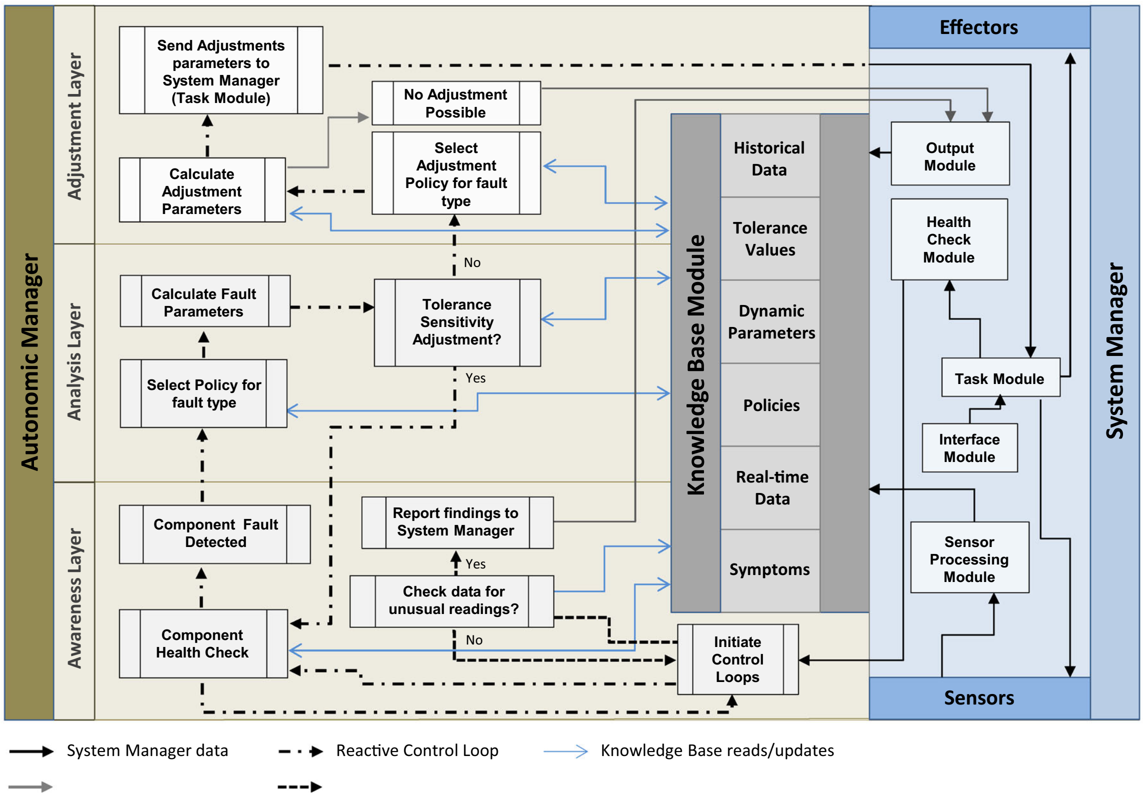

Fig. 21 AIFH autonomic architecture for fault handling in mobile robots

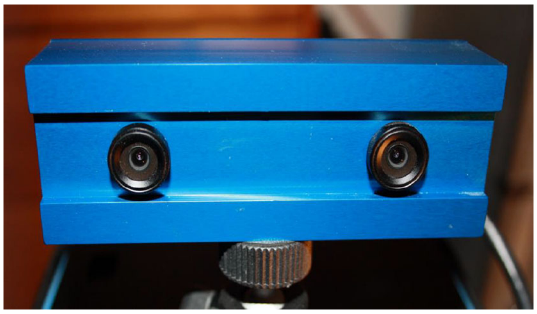

Fig. 22 The PCI nDepth stereo vision camera

ously within the *Reaction loop* (between the Awareness and Analysis layers), therefore creating an infinite fault state.

This architecture attempts to integrate the autonomic principles of *self-healing*, *self-analyzing*, *self-aware*, *self-optimizing*, as described in research by [18,42,43]. 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.

In Sect. 8, we employ another case study (stereo vision camera) and apply the AIFH architectural model to that study. The purpose is to further demonstrate that the AIFH autonomic architecture can apply to other component fault scenarios that can occur within a mobile robot.

8 Experimental validation and analysis (applying the generic autonomic architecture—AIFH)

8.1 Stereo vision camera fault—case study

To evaluate the design of the AIFH architecture (Fig. 21), we have applied it to a further case study centered on hardware faulting within a stereo vision camera sensor. The overall objective was to demonstrate the utility of the generic architecture to a new 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.

8.1.1 Stereo vision camera—properties

The stereo camera can be used to identify obstacles and evaluate their distance from the robot. Figure 22 shows a PCI nDepth^{T}M stereo vision camera. This stereo vision camera and processing PCB board provide depth measurements by using a pair of sensors and a technology called computational stereo vision.

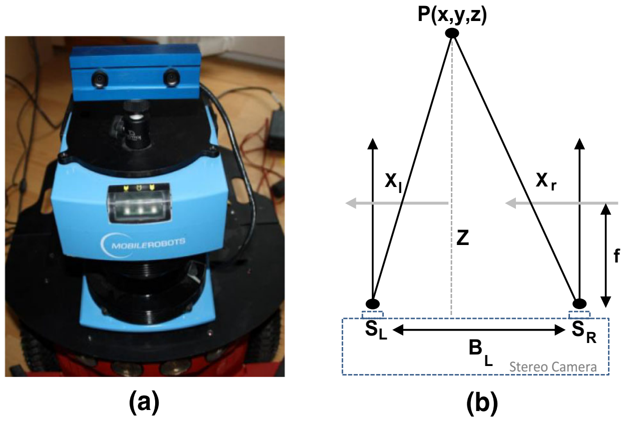

Fig. 23 **a** The PCI nDepth stereo camera mounted on a P3-DX mobile robot. **b** Triangulation method for finding point P

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

8.1.2 Triangulation

Figure 23a shows a PCI nDepth^{T}M stereo vision camera mounted on top of a Pioneer P3-DX. The basis of the technology is that a single physical point in three-dimensional space projects unique images when observed by two separated cameras. Figure 23b 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 6 cm) and **f** represents the focal length of the sensors.

8.1.3 Disparity

This 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 [44].

8.1.4 Awareness (finding a potential fault)

To calculate the distance between the camera and a known object, triangulation stereo vision method can be implemented [45]. 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 24 shows the possible faults that can occur for a sensor in

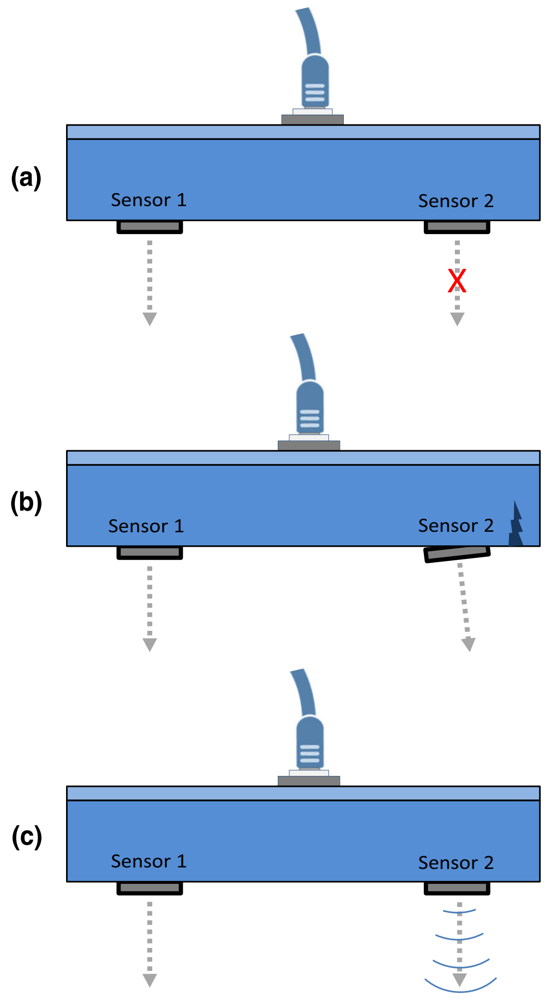

Fig. 24 Stereo vision camera faults. **a** Sensor shutdown, **b** impact (pitch/yaw) and **c** defocus blur

a stereo vision camera setup. If the stereo camera were 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.

In the AIFH architecture (Fig. 21), *Awareness* initiates monitoring and knowledge-based evaluation in order to establish if there is a potential fault with a hardware component. In Fig. 24a, 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 labeled as disabled. In Fig. 24b, 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 [46], 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. These error data are then sent to the Analysis process. Figure 24c shows how defocus blur can potentially influence the quality of the disparity estimate. Research conducted by [47] explains how defocus can lead to objects appearing blurry in the image. Therefore, we can apply equations calculated in [47], 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 Fig. 24b, c will not indicate what sensor has failed. To establish what sensor has failed will require in-depth analysis.

8.1.5 Analyzing (establishing what sensor is faulty)

From the *Awareness* process carried out in Sect. 8.1.3, we need to establish the extent of the fault that has been discovered. The AIFH architecture (Fig. 21) 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 Fig. 24a, 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 Fig. 24b, c, 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 toward 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 25 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 toward 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 Eq. (7), we can calculate distance **d**.

$$d = (\mathrm{ed} + \mathrm{bb}) - \mathrm{sb}. \tag{7}$$

Now, we have established the distance between the stereo camera and the object, and we need to determine what sensor on the camera is faulty. There are several scenarios (Table 7).

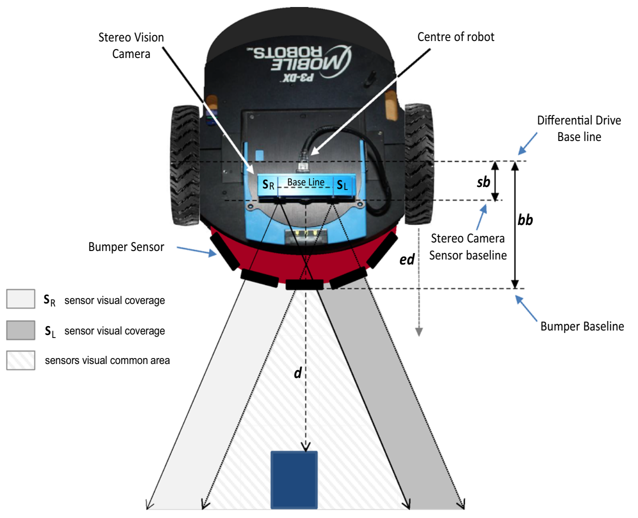
Stereo Vision Camera
Centre of robot
Differential Drive Base line
S_R Base Line S_L
sb
bb
Stereo Camera Sensor baseline
ed
Bumper Sensor
Bumper Baseline
S_R sensor visual coverage
S_L sensor visual coverage
sensors visual common area
d

Fig. 25 The Pioneer P3-DX Bumper can be used to calculate the distance between the stereo camera and the object

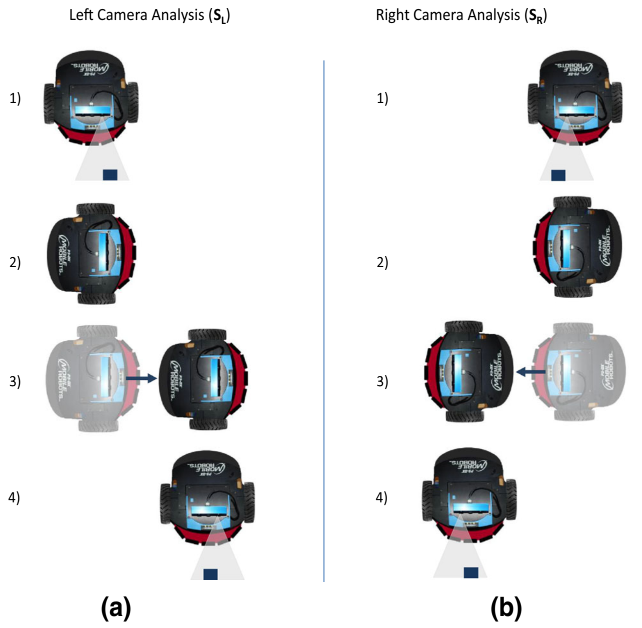

Fig. 26 Representation showing how each camera sensor can be tested by evaluating two images taken by the same camera sensor from its original position and from the position of the opposing camera

From the fault scenario established in Fig. 24b, c, we can assume that both camera sensors are providing data. Therefore, for this experiment we can concentrate on testing scenarios 6–8, in Table 7 only.

To test scenarios (6–8) in Table 7, we must evaluate each camera's sensor individually. The Analyzing procedure carried out by the Autonomic Manager in Fig. 26 involves using a specialized policy to test each individual camera sensor. The procedure involves taking a picture with camera sensor SL (Fig. 25) and then storing these 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 [45] using the stored image and the current image to establish the distance value to the object.

Steps required for sensor evaluation (Fig. 26)

1. Take a picture of the object with one camera only (a) or (b) (Fig. 26) and then store image data.
2. Rotate the robot 90° or −90° depending on what camera sensor is being evaluated (Fig. 26a or b).
3. Move the robot forward a distance equivalent to the stereo camera *Base Line* value between sensor SL and sensor SR (Fig. 25).
4. Rotate the robot 90° or −90° depending on what camera sensor is being evaluated (Fig. 26 (a)) or (b)). Take a picture of the object with the same camera used in (1).

We can then compare the *distance-to-object* result of each camera sensor with that of the known distance value established using the *bumper* sensor [Eq. (7)]. If one of the camera sensor *distance-to-object* results is not within expected tolerances, then we assume this sensor is faulty. If none of the camera sensor *distance-to-object* results are within expected tolerances, then the stereo camera vision device will be

Table 7 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 are unreliable
5	Data (bad)	Disabled	Left camera data are unreliable, right camera is disabled
6	Data (bad)	Data (bad)	Both cameras are providing data that are unreliable
7	Data (good)	Data (bad)	Left camera is providing reliable data, right camera data are unreliable
8	Data (bad)	Data (good)	Left camera data are unreliable, right camera is providing reliable data right

declared as fully disabled and not capable of detecting objects within its path. No adjustment can be made for this scenario.

8.1.6 Adjustment (using one camera sensor as stereo vision camera)

If during the Analysis evaluation, it is established that there is at least one working camera sensor, we can implement a policy to compensate for the other faulty sensor (Algorithm 7). The compensation strategy is like the strategy employed to discover what stereo camera sensor was operating correctly during the analysis stage (Fig. 26).

Algorithm 7: 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
```

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 employed 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.

9 Summary

When operating mobile robots in remote locations, the importance of *self-managed* systems cannot be underestimated, especially when dealing with component failures. With the lack of human intervention available, integrating the autonomic *self** properties [8] can allow a mobile robot to continue to function even while experiencing degrees of hardware failure.

In this paper, we have proposed a generic autonomic architecture (AIFH) for fault management within a mobile robot. We designed a generic architecture by using the data and experiences collected from investigations carried out on individual component failures on mobile robots. As we investigated each case study, we identified common patterns that were required to handle each fault scenario. We adapted the autonomic MAPE-K feedback loop model [8] and the IMD architecture [9], as a foundation, and expanded the design to integrate both the robot System Manager and the Autonomic Manager (Fig. 21). Using the data from previous tasks performed by the robot (i.e., *the knowledge base*), we employed intelligent monitoring to look for subtle changes in performance and thereby identify possible impending faults. There was also a need to revise tolerance values so that the 'awareness' levels were not over-sensitive. We investigated how in-depth analysis can categorize the severity of a fault and therefore take the appropriate action regarding what 'analysis' policy will best suit the situation. With a 'fault' on the robot identified and analyzed, we then implemented a strategy to compensate for the fault and therefore re-establish a level functionality back into the offending component. When the compensation policies were applied to the fault, we analyzed the results and made any necessary adjustments.

Our approach to building an autonomic architecture centered on Awareness, Analysis and Adjustment.

9.1 Awareness

Using current and historical data, we can establish if there is a possible fault within a component. In our research, we discovered various levels of 'awareness', from detecting a fault in a sonar sensor, discovering wheel alignment issues through dead reckoning and predicted faulting due to battery degradation.

9.2 Analysis

When discovering a fault, it is important to establish what extent the fault is affecting the component. In our case study research, analysis on sonar sensor faulting allowed us to establish what sensors were disabled/enabled and how this information affected the ability of the robot to detect objects. Our case study research also revealed the importance of the *knowledge base* ’dynamic parameters’ in providing the Analysis process with current data from components. The Analysis process also provided a means of adjusting *tolerance* values, so that the ‘fault’ reporting did not become over-sensitive within the Awareness layer.

9.3 Adjustment

A major part of the autonomic model is the ability to adapt to changing circumstances. In our research, we created specialized dynamic algorithms that can adapt to various degrees of faulting within a component. In sonar sensor faults, this could either be a simple adjustment for losing one sensor or a complex adjustment were possibly three or more sensors become disabled.

To evaluate the utility of the generic autonomic architecture, we introduced a further case study (Sect. 7). This case study proposed that with the integration of a hardware sensor into the mobile robot, the Autonomic Manager can adapt to handle any possible faults and adjustments that may be required. Each hardware component on the mobile robot requires ‘specialized’ policies for *self-adaption*, but the overall system is designed to handle and process any ‘fault’ situation regardless of the component type. In most cases, the performance of the robot is still impaired (for example, it may take longer to carry out tasks), but the point is that the robot can keep working at a level.

9.4 Why use AIFH?

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

In this work, we researched 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 feedback loop that monitors for faults, whereas the AIFH offers a dual feedback 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 underperforming. The MAPE-K feedback loop is one-way (Analysis leads to Plan, Plan to Execute, etc.); the AIFH (two-way feedback 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 policies 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 with 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 three layers before it can communicate with the *effectors*.

10 Conclusions and future work

The generic autonomic architecture can also be employed in the development of autonomic systems for new robot components. Our work on the generic autonomic architecture has also helped us to better understand how suitable design of future robotic components could greatly facilitate their eventual management within an autonomic framework. We have found in our investigations that certain characteristics in components could be improved to allow flexibility in changing their parameter settings when faced with hardware fault issues, for example, the lack of flexibility in changing individual motor ‘drive’ parameters when dealing with ‘wheel alignment’ issues. Robotic engineers should consider adopting an autonomic approach when designing components and sensors for future mobile robots to make components more adaptable. The AIFH architecture provides a reusable software engineering design, where other developers can reuse.

Future work will concentrate on adapting the generic architecture into a fully implemented system. Further work is required in trying to balance the processing time for both the System Manager and the Autonomic Manager. We would also like to investigate proactive approaches to fault detection, that is, trying to discover early stages of component degradation and therefore alerting the robots System Manager to possible pending faults. This would then allow Users/Mission Control to take appropriate action.

In this journal, we concentrated on handling specific faults, which occurred independently of each other. In future research, handling faults that occur simultaneously would require significant work as a fault within a component may influence how Autonomic Manager can compensate for a fault within a different component. Faults may also occur within the Autonomic Manager itself, but these types of *discreet* faults are beyond the scope of this journal but may be addressed in future work.

References

1. Kawabata K, Akamatsu T, Asama H (2002) A study of self-diagnosis system of an autonomous mobile robot: expansion of state sensory systems. In: International conference on intelligent robots and systems, vol 2, pp 1802–1807
2. Willsky A (1976) A survey of design methods for failure detection in dynamic systems. Automatica 12:601–611
3. Goel P, Dedeoglu G, Roumeliotis SI, Sukhatme GS (2000) Fault detection and identification in a mobile robot using multiple model estimation and neural network. In: Proceedings 2000 ICRA. Millennium conference. IEEE international conference on robotics and automation. Symposia proceedings (Cat. No. 00CH37065), vol 3, pp 2302–2309
4. Abid A, Khan MT, de Silva CW (2015) Fault detection in mobile robots using sensor fusion. In: 10th international conference on computer science education (ICCSE), pp 8–13
5. Zweigle O, Keil B, Wittlinger M, Levi P (2013) Recognizing hardware faults on mobile robots using situation analysis techniques. In: Intelligent autonomous systems 12. Advances in intelligent systems and computing (eds), vol 193 pp 397–409
6. Mendoza JP, Veloso M, Simmons R (2012) Mobile robot fault detection based on redundant information statistics. In: In IROS workshop on safety in human-robot coexistence and interaction
7. Neilson T (2005) Mars exploration rovers surface fault protection. In: Proceedings of the IEEE international conference on systems, man and cybernetics, Waikoloa, pp 14–19. https://doi.org/10.1109/ICSMC.2005.1571115
8. Kephart JO, Chess DM (2003) The vision of autonomic computing. Computer 36(1):41–50
9. Norman DA, Ortony A, Russell DM (2003) Affect and machine design: lessons for the development of autonomous machines. IBM Syst J 42(1):38–44
10. Sterritt R (2005) Autonomic computing. Innov Syst Softw Eng NASA J 1(1):79–88. https://doi.org/10.1007/s11334-005-0001-5
11. Shuaib H, Anthony R, Pelc M (2011) A framework for certifying autonomic computing systems. In: Seventh international conference on autonomic and autonomous systems. Curran associates, pp 122–127
12. Truszkowski W, Hallock H, Rouff C, Karlin J, Rash J, Hinchey M, Sterritt R (2009) Autonomous and autonomic systems: with applications to NASA intelligent spacecraft operations and exploration systems. Ser. NASA Monographs in Systems and Software Engineering, Springer, London. https://books.google.co.uk/books?id=FaFJz_b0fdIC
13. Sterritt R, Hinchey M (2009) Adaptive reflex autonomicity for real-time systems. Innov Syst Softw Eng 5(2):107–115. https://doi.org/10.1007/s11334-009-0090-7
14. Mudgal TR, Kumar S, Chaudhary L (2014) Autonomic computing revolutionizing the way of managing complex systems. Int J Adv Res Comput Sci Softw Eng 4(1):398–402
15. Lewis PR, Platzner M, Rinner B, Tørresen J, Yao X (2016) Self-aware Computing Systems. 1st (ed.) Ser. Natural computing series, Springer, Switzerland, vol 1
16. Jakimovski B, Litza M,Florian M (2006) Development of an organic computing architecture of robot control. In: Informatik 2006 workshop on organic computing
17. Jakimovski B, Meyer B, Maehle E (2009) Self-reconfiguring hexapod robot OSCAR using organically inspired approaches and innovative robot leg amputation mechanism
18. IBM (2005) An architectural blueprint for autonomic computing. vol 7, pp. 1–31
19. Zweigle O, Keil B, Wittlinger M, Haussermann K, Levi P (2013) Recognizing hardware faults on mobile robots using situation analysis techniques. Springer, Berlin, pp 397–409. https://doi.org/10.1007/978-3-642-33926-4_37
20. Khalastchi E, Kalech M, Rokach L, Shicel Y, Bodek G (2012) Sensor fault detection and diagnosis for autonomous systems. In: 23rd International workshop on principles of diagnosis (DX 2012). http://www.ise.bgu.ac.il/faculty/liorr/Eli1.pdf
21. Melchior NA, Smart WD (2004) Autonomic systems for mobile robots. In: International conference on autonomic computing, 2004. Proceedings, pp 280–281
22. Scheutz M, Kramer J (2007) 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
23. De Luna Almeida A, Briot J-P, Aknine S, Guessoum Z, Marin O (2007) Towards autonomic fault-tolerant multi-agent systems. In 2nd Latin American autonomic computing symposium (LAACS'07), Petropolis. https://hal.archives-ouvertes.fr/hal-01305981
24. Crestani D, Godary-Dejean K, Lapierre L (2015) Enhancing fault tolerance of autonomous mobile robots. Robot Auton Syst 68:140–155. https://doi.org/10.1016/j.robot.2014.12.015
25. Lussier B, Chatila R, Guiochet J, Ingrand F, Lampe A, olivier Killijian M, Powell D (2005) Fault tolerance in autonomous systems: how and how much?. In: Proceedings of the 4th IARP/IEEE-RAS/EURON joint workshop on technical challenge for dependable robots in human environments, pp 16–18
26. Bala A, Chana I (2015) Autonomic fault tolerant scheduling approach for scientific workflows in cloud computing. Concurr Eng 23(1):27–39. https://doi.org/10.1177/1063293X14567783
27. Agogino A, Tumer K (2006) Distributed evaluation functions for fault tolerant multi-rover systems. In Proceedings of the 8th annual conference on genetic and evolutionary computation. ser. GECCO '06, New York, pp 1079–1086. http://doi.acm.org/10.1145/1143997.1144171
28. Krzyzanowski P (2009) Fault tolerance—dealing with an imperfect world. https://www.cs.rutgers.edu/~pxk/rutgers/notes/content/ft.html
29. David L (2014) How wheel damage affects mars rover curiosity's mission. https://www.space.com/26472-mars-rover-curiosity-wheel-damage.html
30. McKee M (2006) Mars rover's broken wheel is beyond repair. https://www.newscientist.com/article/dn8944-mars-rovers-broken-wheel-is-beyond-repair/
31. Lawton J (2018) Rass: resilient autonomic software systems. https://cs.gmu.edu/~menasce/rass/
32. Sterritt R (2007) Systems and software engineering of autonomic and autonomous systems. Innov Syst Softw Eng 3(1):1–2. https://doi.org/10.1007/s11334-006-0018-4
33. Doran M, Sterritt R, Wilkie G (2016) Autonomic self adaptive robot wheel alignment. In: Adaptive 2016: the eighth international conference on adaptive and self-adaptive systems and applications IARIA, pp 27–33

34. LMS 200 / LMS 211 / LMS 220 / LMS 221 / LMS 291 Laser Measurement Systems, SICK AG, June 2003, https://www.sick.com/de/en/detection-and-ranging-solutions/2d-lidar-sensors/c/g91900
35. Doran M, Sterritt R, Wilkie G (2017) Autonomic sonar sensor fault manager for mobile robots. World Acad Sci Eng Technol Int J Comput Inf Eng 11(5):603–610
36. Ritchie AG, Lakeman B, Burr P, Carter P, Barnes PN, Bowles P (2001) Battery degradation and ageing. Springer, Boston, pp 523–527. https://doi.org/10.1007/978-1-4615-1215-8_58
37. Doran M, Sterritt R, Wilkie G (2018) Autonomic management for mobile robot battery degradation. World Acad Sci Eng Technol Int J Comput Inf Eng, Forthcoming 2018
38. LTD YB (2008) Np series np7.5-12 data sheet. http://www.yuasabatteries.com/np_industrial_literature.php
39. Mei Y, Lu Y-H, Hu YC, Lee CSG (2005) 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
40. Omron Adept MobileRobots L (2007) Pioneer 3 operations manual, version 5. http://www.mobilerobots.com/ResearchRobots/PioneerP3DX.aspx
41. Sylla AN, Louvel M, Rutten E, Delaval G (2017) Design framework for reliable multiple autonomic loops in smart environments. In: 2017 International conference on cloud and autonomic computing (ICCAC), pp 131–142
42. Poslad S (2009) Autonomous systems and artificial life ch 10. Wiley , pp 317–341. https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470779446-ch10
43. Bertels K, Nami MR (2007) A survey of autonomic computing systems. in Autonomic and autonomous systems, international conference on (ICAS), vol 00, p 26. https://doi.org/10.1109/CONIELECOMP.2007.48
44. Raajan N, Ramkumar M, Monisha B, Jaiseeli C, venkatesan SP (2012) Disparity estimation from stereo images. In: International conference on modelling optimization and computing, Springer, vol 38, pp 462–472. http://www.sciencedirect.com/science/article/pii/S187770581201-9704
45. Song W, Xiong G, Cao L, Jiang Y (2011) Depth calculation and object detection using stereo vision with subpixel disparity and hog feature. Advances in information technology and education. Springer, Berlin, pp 489–494
46. Zhao W, Nandhakumar N (1996) Effects of camera alignment errors on stereoscopic depth estimates. Pattern Recognit 29(12):2115–2126 http://www.sciencedirect.com/science/article/pii/S003132039600-0519
47. Yang CC, Huang SK, Shih KT, Chen HH (2018) Analysis of disparity error for stereo autofocus. IEEE Trans Image Process 27(4):1575–1585