A Multi-Agent System for Autonomous Adaptive Control of a Flapping-Wing Micro Air Vehicle

Michal Podhradsky
Dept. of Elec. & Comp. Engr.
Portland State University
Portland, OR 97207

Garrison Greenwood
Dept. of Elec. & Comp. Engr.
Portland State University
Portland, OR 97207

John Gallagher
Dept. of Comp. Sci. & Engr.
Wright State University
Dayton, OH 45435

Eric Matson
Dept. of Comp. & Info. Tech.
Purdue University
West Laffaette, IN 47907

***Abstract*—Biomimetic flapping wing vehicles have attracted recent interest because of their numerous potential military and civilian applications. In this paper we describe the design of a multi-agent adaptive controller for such a vehicle. This controller is responsible for estimating the vehicle pose (position and orientation) and then generating four parameters needed for split-cycle control of wing movements to correct pose errors. These parameters are produced via a subsumption architecture rule base. The control strategy is fault tolerant. Using an online learning process an agent continuously monitors the vehicle's behavior and initiates diagnostics if the behavior has degraded. This agent can then autonomously adapt the rule base if necessary. Each rule base is constructed using a combination of extrinsic and intrinsic evolution. Details on the vehicle, the multi-agent system architecture, agent task scheduling, rule base design, and vehicle control are provided.**

I. Introduction

Biomimetic flapping-wing vehicles have been the focus of much recent research due to their potential for both civilian and military application. In either potential application area, adaptive, fault-tolerant, control is paramount. This paper describes a multi-agent system for adaptive control of a model biomimetic flapping-wing vehicle. The vehicle employed in this research is a hardware analogue of a minimally-actuated flapping-wing vehicle introduced by Wood [1], [2] with core control laws introduced and subsequently refined by Doman et al. [3], [4]. The analogue vehicle [5]–[7] operates similarly to the minimally actuated vehicles considered by Wood and Doman et al. in that all propulsion and control are provided by two minimally actuated wings, each of which possess a single active and a single passive degree of freedom. It differs in that the analogue vehicle's active degrees of freedom are driven by a DC motor through a four-bar linkage instead of a piezoelectric transducer and in that the analogue vehicle is mounted vertically on a circular puck and supported from below by either an air or fluid cushion. These changes allow us to experiment with the control of translation and roll without need to address the practical difficulties of balancing generated lift and vehicle weight.

Previous work employed variants of the controllers discussed in [3], [4] augmented with adaptive wing beat oscillators [8]–[10] that provided adaptation at the inner-most layer of vehicle control (wing flapping patterns). The goal of that work was to provide adaptation not by changing control laws that related desired forces and torques to stereotyped wing motions, but rather, to change the wing stereotyped motions to adapt generated forces to the needs of control laws designed for undamaged wings. In other words, the salient adaptation was that damaged wings learned to move in ways that mimicked the force and torque generation of undamaged wings.

In contrast to that work, this paper presents a design for a multi-agent system (MAS) where control laws are directly adapted at a higher level of abstraction in the control law hierarchy. In this system, agents are responsible for collecting and estimating vehicle pose, recording waypoint locations for trajectory following, generating inputs needed by the split-cycle oscillator, monitoring vehicle behavior and, when necessary, conducting diagnostics and adapting the control rule base. The initial set of control laws are designed using a combination of extrinsic and intrinsic evolution [11]. The ultimate vision is that both forms of adaptation co-exist in the vehicle so that the benefits of each approach are equally available.

The paper is organized as follows. A basic overview of our flapping wing vehicle is described in the next section. Section III describes the multi-agent architecture while Section IV gives implementation details. The online learning algorithms and rulebase adaption methods are described in Section V. Part of the design requires use simulation tools, which are discussed in Section VI. Finally, our next steps for full implementation on the hardware are given in Section VII.

II. Vehicle and Environment Description

A. Vehicle Configuration

A conceptual vehicle closely related to those described by Wood and Doman et. al. is presented in [8]. The physical analogue descendant from it is described in [5]–[7]. Both vehicles operate in a qualitatively similar manner with two minimally actuated wings providing all propulsion and control forces. They differ only in scale and in that the physical vehicle is supported from below by a fluid or air cushion. The physical vehicle, with its externally provided lift support, approximates a passively upright-stable version of the vehicle in [8] operating near its hover wing flapping frequency. Both vehicle types (hereafter referred to as "the vehicle") have two wings mounted in the X_b–Y_b body plane (see Figure 1). These

978-1-4799-7560-0/15/$31 ©2015 IEEE

978-1-4799-7560-0/15 $31.00 © 2015 IEEE
DOI 10.1109/SSCI.2015.154

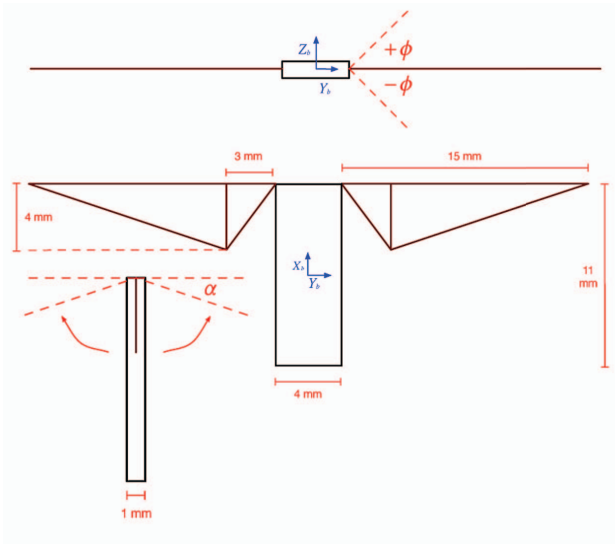

Fig. 1. Orthographic view of flapping wing vehicle [8]. Both wing spars are restricted to rotational motion about their joints with the body and in the Y_b–Z_b plane. The range of those rotations is $[-1 \ldots 1]$ radians, α is between $\pi/6$ and $\pi/2$ radians. Note that the dimensions are for orientation purposes only, and differ on the actual vehicle.

wings are actively actuated within the range of $\pm\phi$. As the spars rotate, dynamic air pressure lifts the triangular wing planforms (membranes) up to an angle of α radians under a base vector embedded in the Y_b–Z_b plane. Individual wing flaps produce independent lift and drag forces at each of the two wing roots (points of attachment of the wings to the body). These can be resolved into body frame forces and torques and cause changes in the whole vehicle's position and pose in three space.

B. Cycle Averaged / Split Cycle Control

In cycle averaged control, one bases vehicle control on estimates of what forces a wing would produce, on average, over a single wing beat. For example, a cycle averaged altitude controller might compute the error between current and desired altitude, use a error feedback control law to compute a desired force to apply to the body, and finally use a model of the vehicle?s wings to compute the parameters of a single wing beat that, when adopted by both wings, produces the desired force (on average) during the whole wing beat. Cycle-averaged control wraps a feedback control law around whole wing beats as atomic constructs rather than around finer-scaled micro-motions of wings. The desired wing motions are ?communicate? to the wings once per wing beat as a small number of shape parameters that define how the wing will move during that wing beat.

Split-cycle control is a special case of cycle-averaged control in which wing beats are composed of two half-cosine waves, one each to govern the wing?s upstroke (front to back) and downstroke (back to front). The shape parameters communicated to each wing are a flapping frequency (ω) and an upstroke/downstroke transition parameter (δ). Advancing the upstroke (and consequently impeding the downstroke) produces a forward force while keeping the wing beat frequency constant. Formally, $\phi_U = \cos((\omega - \delta)t)$ and $\phi_D = \cos((\omega + \sigma)t)$ where σ is dependent on δ. From [4] we know that $\delta \in [-\infty \ldots \omega/2]$ although certain value ranges are particularly important. If $\delta = 0$ the upstroke is symmetrical to the downstroke and a regular wingbeat occurs. However, if $\delta > 0$ the upstroke is impeded and the downstroke is advanced, as shown in Figure 2. As a result, a force is generated in direction of the downstroke. Conversely if $\delta < 0$ then the downstroke is impeded and the upstroke is advanced, as shown in Figure 3 , resulting in a force in the direction of the upstroke. These lateral forces act on the vehicle?s body via a moment arm producing an angular momentum. Put simply, by applying the split cycle the vehicle can turn. (See [4] for a derivation and proof of split-cycle operation.)

A conventional application of split-cycle control to this vehicle might entail an "outer loop" of multiple body axis controllers (e.g., one that computes altitude error and determines a flapping frequency for the wings, one that computes a roll axis angle error and computes antagonistic δ shifts to produce a roll moment, etc.), and an allocator that harmonizes all of the flapping frequency and δ shift commands made by the various axis controllers for presentation to an "inner loop" controller that would ensure the wings follow the correct cycle averaged trajectories. Control would then consist of an outer loop that, based on vehicle state, provides ωs and δs that should produce forces required to effectively correct position and pose errors and an inner loop that, receiving those δs and ωs, would ensure the wings moved as required.

Our previously described attempts at controller adaptation have occurred in the inner loop. This paper presents an architecture to enable co-adaptation in the outer loop. Although in theory a full 6DOF control can be achieved using split cycle with only minor modification (see [4] for details), our current work will be constrained to 3DOF movement in two dimensions, primarily because the weight of the mechanical components makes takeoff impractical for our mechanical analog. Nonetheless, the developed control and fault recovery algorithms are expected to remain valid in three dimensions and can be tested in simulation or in subsequent free-flying vehicles.

C. Experimental Setup and Environment

All experiments are to be conducted in a large ($5' \times 5'$) water tank. The vehicle currently can move on a two dimensional plane and rotate around its X_b axis. The vehicle is equipped with a pair of Lithium Polymer batteries, a power distribution board and a main computer, as shown in Figure 4. All hardware is mounted on a carbon-fiber platform, attached to a floating Styrofoam puck. The water surface acts as a mechanical low-pass filter, slowing down the vehicle movement and dampening disturbances. A camera is placed above the water tank, such as its field-of-view encompasses the entire water tank. The camera locates and records the vehicle position. The vehicle has color markers for this purpose. The

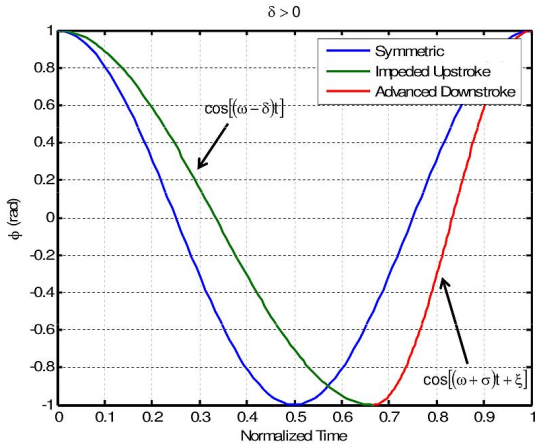

Fig. 2. Split-cycle results for $\delta > 0$

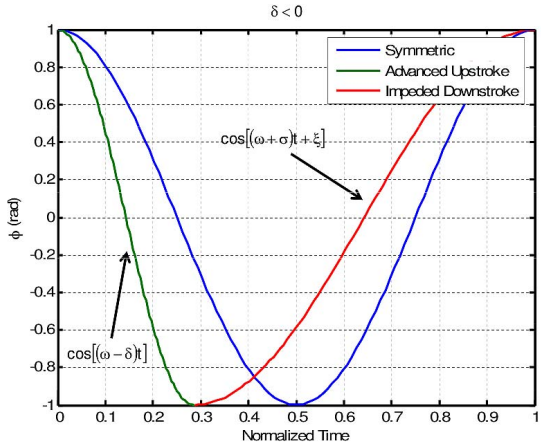

Fig. 3. Split-cycle results for $\delta < 0$

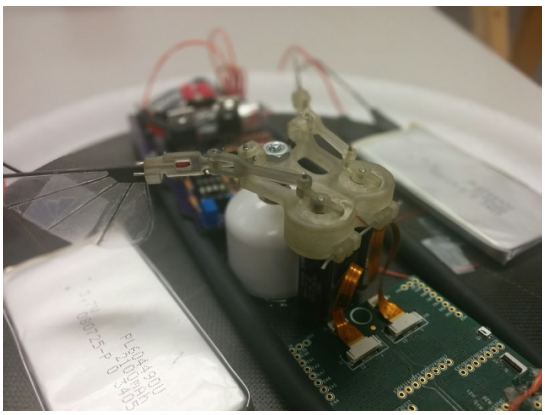

Fig. 4. Assembled vehicle—note wings in the middle, LiPo batteries on sides, the power distribution board in the back and the control board in front.

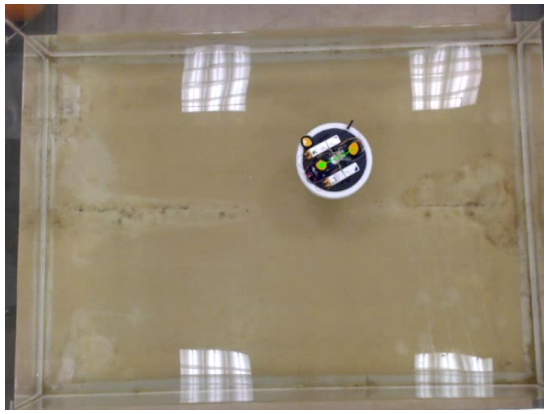

Fig. 5. Vehicle during an experiment in the water tank (top-view). Note the color markers used for machine vision pose estimation.

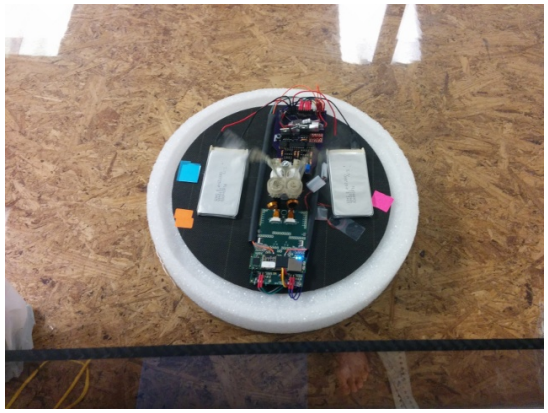

Fig. 6. Vehicle during an experiment in the water tank (close-up view). Note the color markers used for machine vision pose estimation.

experiment setup as seen from camera is shown in Figure 5. Figure 6 shows a close-up view of the vehicle, including the color markers. The video stream from the camera is processed on a regular laptop computer, using the *OpenCV* library for computer vision. The processed video is recorded for reference, and the estimated pose is sent to the onboard MAS via a WiFi link.

III. Multi-Agent Architecture

In this section we describe the multi agent control architecture and the various agents involved in the vehicle control. The vehicle is assumed to be moving in an enclosed, wind-free environment with no obstacles (other than boundaries). It is required to follow a set of predefined trajectories specified by a sequence of waypoints.

Practically, this is achieved by equipping the vehicle with a floating support and placing it in a large water tank. The boundaries of the perimeter are the walls of the water tank. An overhanging camera is used to estimate the pose (position and orientation) of the vehicle. A software simulation of the vehicle is also available. The simulator is further described in Section VI.

A. Agent Description

The MAS consists of five agents. A *collection agent* receives pose information x, y, ψ from the camera (or simulator), and runs smoothing and averaging algorithms to compute the estimated pose x', y', ψ'. A *monitor agent* observes vehicle behavior and requests vehicle diagnostics if the behavior has deteriorated too much. The *strategy agent* keeps a list of desired waypoints, and provides them upon request to a *controller agent*. The controller agent determines the split-cycle oscillator control inputs (a δ and ω for each wing) based on the vehicle pose. Finally, a *diagnostic agent* runs vehicle diagnostics and determines if a fault occurred and ultimately decides whether the controller agent's rulebase has to be adapted. The MAS diagram is shown in Figure 7. Each agent is further described below.

1) Collection Agent:

Precepts: x, y, ψ
Outputs: x', y', ψ'
Tasks: Collects high-speed data x, y, ψ from either simulator or the camera, and computes smoothing and averaging of the data. Outputs estimated pose data x', y', ψ' at a lower rate. There are several filtering options under consideration, e.g. an exponential moving average filter.

2) Monitor Agent:

Precepts: x', y', ψ', active rule from rulebase
Outputs: issue diagnostics command
Tasks: Receives estimated position data from the collection agent, and movement being executed from the controller agent. Monitors performance of the vehicle, and in case of insufficient results, directs diagnostics agent to run diagnostics.

3) Strategy Agent:

Precepts: none
Outputs: W_j
Tasks: Stores list of waypoints. Sends next waypoint W_j upon request.

4) Controller Agent:

Precepts: x', y', ψ', W_j
Outputs: $\delta_L, \delta_R, \omega_L, \omega_R$
Functions: Consists of one or more agents. Responsible for determining control inputs $\delta_L, \delta_R, \omega_L, \omega_R$ based on the vehicle pose x', y', ψ' and the desired position W_j. When the vehicle reaches a waypoint, requests new waypoint coordinates from the strategy agent. Runs maneuvers for diagnostics testing.

5) Diagnostic Agent:

Precepts: x', y', ψ', W_j
Outputs: computes likelihood of fault
Functions: Initiates diagnostics testing. Computes likelihood of failure. Determines if behavior must be adapted to compensate for faults.

B. Agent Scheduling

The scheduler is responsible for specifying when individual agents execute their assigned tasks. In this application scheduling is event oriented rather than time oriented. Our scheduler is loosely based on the scheduler used in the RePast agent-based toolkit [12].

The scheduling process can be abstractly thought of as a sequence of hooks on a wall (see Fig 8). An agent is "hung" on a hook if it is supposed to execute some task at some future time. However, time is relative, not absolute—i.e., hooks are not associated with some specific time nor does hook spacing reflect time intervals; hooks merely establish a partial ordering of agent tasks. For example, if agents $x, y,$ and z are hung on hooks 3, 4 and 7 respectively, this simply says agent x performs some task before agent y which performs some task before agent z. Hooks therefore just order agent behaviors with respect to each other. Only one agent can be hung on a particular hook because agents do not execute tasks concurrently. We say an agent is *stepped* if it is directed to perform some action or task.

The entire scheduling process is event driven. A first-in-first-out (FIFO) buffer is used as an event queue. As agents perform perform assigned tasks—i.e., they are stepped by the scheduler—they may post events in the FIFO, which might cause other agents to be stepped at a later time. Events have a header and a body. The header tells which agent posted the event, the event type and possibly a timestamp. The body contains attributes unique to the event.

The scheduler unloads the buffer, analyses the events, and processes events by stepping a specific agent. Events are pulled from the FIFO as quickly as possible but stepping agents is deferred while the FIFO is not empty. All the scheduler does at this time is decide upon which hook to hang the event. Some shuffling of agents already hung on hooks may occur depending on the priority of the event. When the FIFO is empty the scheduler starts pulling agents off of hooks in a "first-hung-first-pulled" order and steps them. A stepped agent may be provided information from the body of the event that prompted it being stepped.

A simple example will help fix ideas. Suppose the controller agent has just output new δ and ω values. This agent would post a "new δ/ω output" event in the event queue. The body of this event would identify which rule in the rulebase fired so the commanded vehicle movement is recorded. When offloaded from the FIFO the scheduler would hang a monitor agent tag on a hook along with the rule ID extracted from the event body. Once the FIFO is empty the scheduler pulls the monitor agent tag off of the hook and steps the monitor agent to commence observing the vehicle movement. The monitor agent would be informed at that time which rule fired.

IV. Agent Implementation

This section describes each of the aforementioned agents in more depth, including the implementation details.

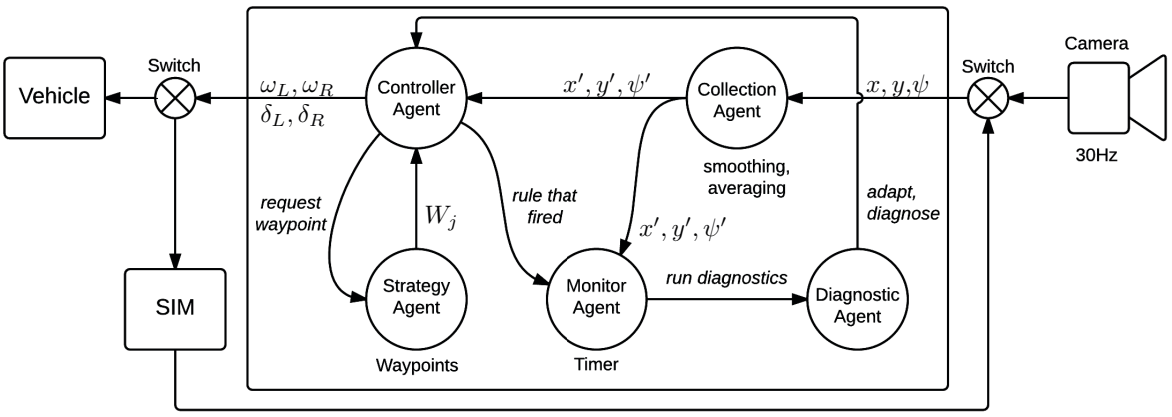

Fig. 7. Diagram of Agent-based control architecture. Detailed information about each agent is provided in Section III-A. More info about the simulator can be found in Section VI. Note that the arrows between the agents do not reflect inter-agent messages—the communication is done by event processing, as described in Section III-B

Fig. 8. An example showing agents hanging on hooks waiting to be stepped by the scheduler. The left-to-right order reflects the relative ordering of the agent stepping.

A. Controller Agents

These agents compute the δ_L and δ_R values needed to move the vehicle between waypoints. The vehicle will typically have to adjust its course as it moves along a trajectory. However, that does not mean new δ values are needed at the beginning of each wing beat.

There are 4 control inputs to the vehicle available (specifically $\delta_L, \delta_R, \omega_L, \omega_R$), but only two actuators (the left and right wing) thus the control inputs are not independent. This situation can be described as *under-actuated* vehicle, which means we cannot control the position and course independently, and that poses a greater challenge for the controller agents. For example, if the vehicle is moving forward and is heading slightly left off the desired course, it can't turn right without affecting the forward motion.

An arbitrary position and orientation can be achieved by a combination of linear motion (forward/backward movement) and rotation of the robot. Forward motion is done by increasing both δ_L and δ_R, while rotation is achieved by increasing δ_L and decreasing δ_R and vice versa. Note that although backward movement is possible, it is not used in this case. Required movements are summarized in Table I.

The vehicle is required to make two distinct turns—"hard" turn—a rotation of $90\,\mathrm{deg}$ and is used for evasive maneuvers; and "partial" turn, used for slight course corrections. Each turn requires different values of δ_L, δ_R, but the direction of change in δ is the same for both turns. The values of δ will be stored in a look-up table. Note that increasing ω_L or ω_R (while keeping the same relative value of δ) creates stronger moments and forces, which might be necessary for faster movement, especially rotation. Suitable values for ω_L and ω_R have yet to be determined.

The vehicle must make specific movements to follow a trajectory and a rulebase determines which movements will be needed. These rules are organized in a *subsumption architecture* [13]. Under this architecture all control rules have an IF-THEN syntax. The rulebase consists of the rules shown in Table II. Each movement indicated in a rule's consequent has an associated set of δ and ω values, which are extracted via a table lookup.

A subsumption architecture consists of a series of layers where lower layer rules produce simple, critical behavior such as avoiding obstacles while higher levels produce more sophisticated behavior needed for trajectory following. Higher level behavior subsumes lower level behavior. A subsumption architecture is ideal for navigation control in dynamic physical environments. It permits reactive behavior without resorting to prior path planning because there is no world model required.

Layer 1 has the highest priority. If the vehicle reaches the borders of the perimeter (in this case walls of the water tank), it will turn right to avoid the obstacle. The second highest priority detects whether the vehicle reached the desired waypoint W_j[1]. At that time the controller agent acquires the coordinates of the next waypoint on the trajectory W_{j+1} from the strategy agent.

[1]The vehicle reaches waypoint W_j if it is within distance ϵ of that waypoint.

Movement	Control Input
Move Forward	$\delta_L \uparrow$ & $\delta_R \uparrow$ (identical)
Left Turn	$\delta_L \downarrow$ & $\delta_R \uparrow$ (opposite)
Right Turn	$\delta_L \uparrow$ & $\delta_R \downarrow$ (opposite)
Idle	$\delta_L = \delta_R = 0$
Increase velocity	$\omega \uparrow$
Decrease velocity	$\omega \downarrow$

TABLE I
REQUIRED VEHICLE MOVEMENTS. $\uparrow, \downarrow$ INDICATE DIRECTION OF CHANGE, NOT ITS MAGNITUDE. SEE SECTION IV-A FOR MORE DETAILS.

Layer	Behavior
6	**if** true **then** *Idle*
5	**if** heading left **then** *Partial right turn*
4	**if** heading right **then** *Partial left turn*
3	**if** heading at waypoint **then** *Move forward*
2	**if** at waypoint W_j **then** Get new waypoint W_{j+1}
1	**if** outside perimeter **then** *Hard right turn*

TABLE II
SCHEME OF CONTROLLER AGENT SUBSUMPTION ARCHITECTURE, LAYER 1 HAS THE HIGHEST PRIORITY.

A new waypoint should be requested in a timely manner to prevent vehicle from unnecessary course corrections and large control actions. The third layer ensures that if the vehicle is pointing at the waypoint, it will move towards the waypoint. If it is not pointing in the right direction, layers 4 and 5 will turn the vehicle until it is pointing right at the waypoint, so the third layer (i.e. "move forward") takes control. Finally, if there is nothing better to do, the vehicle idles at its current location until it gets new commands.

B. Monitor Agent

This agent is responsible for monitoring the vehicle's performance. During normal vehicle operation the controller agent posts an event every time a new δ and/or ω value is sent to the split-cycle oscillator. That event identifies the specific rule that fired so the monitor agent knows the expected movement (e.g., *hard right turn*) and, when stepped by the scheduler, begins tracking the movement. If the vehicle's movement doesn't match the expected movement, the monitor agent will post a "poor performance" event in the event queue. No event is posted if the behavior is okay. The scheduler will process this poor performance event by stepping the diagnostic agent to run diagnostics.

C. Diagnostic Agents

This agent assesses the vehicle's reliability. The vehicle has no specific fault detection and isolation capability. It is worth noting that under such circumstances there is, from a behavioral standpoint, no real difference between a vehicle mechanical failure or a sensor failure since both produce the same outcome—an inability to follow a desired trajectory. Nevertheless, rather simple diagnostic routines can identify degrading behavior even if the exact cause is not knowable.

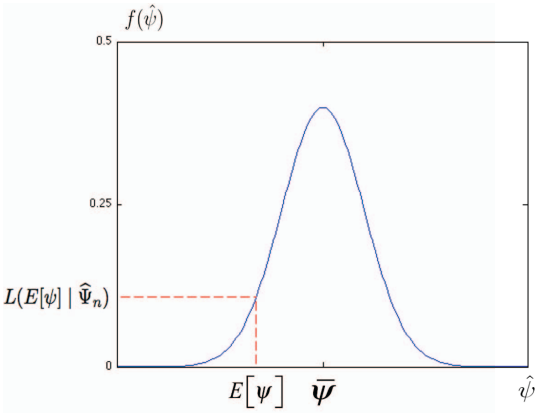

Fig. 9. Calculation of likelihood of getting the expected pose from a pose density function centered at the sample mean of the n rotation estimate data samples $\hat{\Psi}_n$

Diagnostics could be performed at regular intervals. For example, every say 5 minutes of flight time the vehicle could temporarily idle and then quickly run the diagnostics. However, a more practical approach is to exploit the online learning performed by the monitor agent. The monitor agent will trigger the diagnostics if and only if a degraded performance is observed.

Faults are detected using a Bayesian type of behavior monitor where *likelihood functions* give a qualitative measure of maneuver capability. The idea behind diagnostics is simple: command the vehicle to perform some maneuver and see if can do it within a prescribed time frame. A rotation through some angle—e.g. $\pi/2$ radians—is a simple and non-trivial maneuver since it requires $\delta_L \neq \delta_R$. The precise δ values are stored in a table as described previously. Note that a complete diagnostic would required the vehicle to rotate in both directions. Pose samples $\hat{\Psi}_n = (\hat{\psi}_1, \hat{\psi}_2, \ldots, \hat{\psi}_n)$ can be recorded by diagnostic agents over some time window and the associated sample mean and sample variance are easily computed. This information is sufficient to construct a Gaussian *pose density function*

$$f(\hat{\psi}) = \frac{1}{\sigma_\psi \sqrt{2\pi}} \exp\left(-\frac{(\hat{\psi} - \overline{\psi})^2}{2\sigma_\psi^2}\right) \tag{1}$$

where $\overline{\psi}$ is the sample mean and σ_ψ^2 is the sample variance.

The control agent outputs a specific δ_L and a δ_R, which are expected to produce some change in pose. Thus the control agent has some expected pose movement $E[\psi]$ in mind. A diagnostic agent uses the pose density function $f(\hat{\psi})$ to compute the likelihood of $E[\psi]$ given the pose data samples $\hat{\Psi}_n$. This concept is illustrated in Figure 9.

A high likelihood suggests the vehicle mechanical hardware and sensor hardware are operating normally while a low likelihood indicates something is wrong. Dividing the likelihood function value by the sample mean (or equivalently just ignoring the $1/(\sigma_\psi\sqrt{2\pi})$ coefficient in Eq. 1) makes $L(E[\psi]|\hat{\Psi}) \in [0,1]$. Then 'high' and 'low' likelihoods are defined by a threshold parameter λ on the unit interval. That is, $L(\cdot) < \lambda$ indicates a low likelihood the vehicle can maneuver properly and some corrective action is required

The only possible recovery mechanism is to adapt the vehicle's behavior by modifying the rules in the rulebase. The next section describes how this adaption is done.

V. Agent Online Learning

There are two times when online learning is required. Learning is used to identify appropriate δ and ω values needed for control of the vehicle. This section discusses the details about the learning process.

A. Initial Learning

Every vehicle is slightly different due to inherent nonlinearities such as slip between linkages. Thus the same δ and ω values cannot be used for every vehicle; they must be learned. First the vehicle learns values needed to execute the basic movements in Table I. These values are then linked to rule consequents from Table II. The δ and ω values will be determined during this initial learning phase using a combination of extrinsic and intrinsic evolution [11].

The needed parameters will be evolved using a (1, 10)-ES. The genotype is

$$\{\delta_L, \delta_R, \omega_L, \omega_R \ ; \ \sigma_1, \sigma_2, \sigma_3, \sigma_4\}$$

where the first 4 parameters are object parameters and the second 4 parameters are strategy parameters used to control the mutation step size. Most likely a linear reduction schedule will be sufficient for adapting the strategy parameters. The vehicle will be placed in its operational environment (a water tank) and object parameter values will be intrinsically evolved for each movement. (Initial object parameter values will be evolved extrinsically using an in-house developed simulator.) Intrinsic evolution will be run with on-board computing resources.

The monitor agent observes the behavior of each evolved object parameter set and terminates the evolutionary algorithm for a specific rule when acceptable behavior is achieved. The goal here is not to achieve optimal movements but rather smooth and repeatable correct movements. Thus fitness will be computed from the average behavior over a small number of trials. After learning is completed, the evolved values will be stored in a library (i.e. a look-up table). Once all rules are evolved the vehicle is ready for trajectory following.

B. In-Flight Learning

Rules learned during the initial learning phase perform well during a normal flight, but in the presence of faults it will be necessary to adapt them. This online learning phase is performed continuously. Each time a rule fires the monitor agent is informed so it knows what maneuver was commanded. The monitor agent observes the x', y', ψ' pose parameters and determines if the performance is within limits or is degrading. Thus the monitor agent continuously learns about how the vehicle is performing. If the observed behavior deviates too much from the expected behavior then diagnostics are run. If the diagnostics confirm the behavior has degraded below some threshold then the rulebase is adapted. Adaption, described below, entails modifying rules' consequents in order to restore vehicle's functionality.

C. Rule-base Adaptation

Given size and weight restrictions, which don't allow us to use conventional fault recovery methods such as redundant hardware, it is not possible to recover from every possible fault. We therefore restrict the adaption to cover only a small subset of predefined faults. The recovery mechanism relies on adapting new consequents for rules from Table II, so that the desired motion (i.e. *Partial left turn*) is still achievable. These new δ and ω values will be intrinsically evolved just like the initial online learning was done. The question is how do we intrinsically evolve new parameters for specific faults?

We will borrow concepts used in conventional *failure modes & effects testing* (FMET). In this type of testing a set of predefined faults is inserted into the system under test one at a time and their effect is observed. This testing is always conducted in a laboratory environment where the effects are closely monitored and controlled to prevent damage to the system. In this particular research effort "faults" such as using different wing sizes or a stuck linkage will be intentionally put into the vehicle and an evolutionary algorithm will then evolve new parameter values. As before, this evolutionary algorithm will run using onboard hardware resources. The evolved values will be added to the rulebase library.

Under normal operation a single set of $\delta_L, \delta_R, \omega_L$ and ω_R values is sufficient to perform the maneuvers shown in Table I. However, under faults most likely a sequence of δ and ω values will be required, with a new set generated every few hundred wingbeats (because of the slow vehicle dynamics). This sequence of values can be intrinsically evolved too. In this case the genotype described earlier must be expanded to handle multiple δ and ω values for each maneuver.

Of course prior to initiating fault recovery operations *fault detection & isolation* (FDI) must be done. This process detects if a fault exists and tries to isolate it to a specific subsystem or component. Since the ability to recover from faults on the vehicle is severely limited, isolation is unnecessary. Fortunately detection is rather straightforward. As stated above, the monitor agent is continuously learning about the vehicle capabilities; it can therefore detect poor performance by comparing observed behavior against expected behavior. If the behavior is poor it would post an event in the event queue. The scheduler would then step the diagnostic agent to start diagnostics. Diagnostics should be able to confirm both degraded performance and identify which of the pre-defined faults is present.

The abstract sequence of commands used during FDI is as follows:

1) Monitor Agent (MA) detects unsatisfactory performance and posts an event in event queue
2) Diagnostics Agent (DA) is scheduled to perform diagnostics
3) DA posts a diagnostic event. Controller Agent (CA) stops waypoint following and enters diagnostics mode
4) CA starts the test manuever
5) MA monitors test progress and passes results to DA
6) DA computes the likelihood of a fault
7) If DA detects no problem, it posts a normal operation event. CA resumes waypoint following
8) If DA detects a problem, it posts a faulty operation event. CA extracts new rulebase from library.

Essentially we will build a library of rules for both the fault-free and the faulty vehicle. Once FDI is finished fault recovery begins. Recovery only requires replacing the current controller agent's rulebase with the appropriate pre-stored rulebase associated with the identified fault from the library. Thus fault recovery can be done very quickly.

VI. Simulation

A simulator of the vehicle is used for extrinsic evolution during initial learning (see Section V-A for details). The simulator contains a simplified model of the vehicle with 3DOF. A simplified model assumes no external disturbances, such as wind. Also assumes a perfect control over wing position (i.e. no slip in the linkages). It assumes a perfectly balanced vehicle, moving on a frictionless plane.

The simulator calculates cycle averaged lift and drag forces, meaning the produced lift and drag forces are averaged over one full wingbeat. The produced forces are then propagated to the body model, creating moments and change in orientation and position of the vehicle. Cycle averaged forces are a good approximation, because the low level controller cannot change the δ and ω parameters more often than at the beginning of the wing beat.

As mentioned previously, the simulator is not intended to perfectly model the vehicle, but to provide a reasonably accurate initial values for the learning algorithms and their verification.

The core of the simulator is a Java library containing vehicle dynamics, and performing all necessary lift and drag calculations. The library was developed in the research group of one of the authors of this paper. The agents are implemented using the MASON toolkit [14], which will also be used as a visualization front end. Initially the MASON toolkit will be used to verify the MAS architecture—e.g. to make sure that the correct rules from the rulebase are firing in right order, and to check the correctness of the scheduler by moving the vehicle along predefined test trajectories.

VII. Final Remarks

The design of the MAS controller is complete and incorporating it into the vehicle hardware will be straightforward. The most difficult aspect of testing will probably be the diagnostic agent checkout because autonomous fault detection is not a trivial task. As stated previously we will inject faults into the vehicle and then adapt the MAS behavior by replacing its rulebase. This replacement rulebase will mostly be a sequence of δ and ω values. At this time we do not know how long that sequence should be nor how often to apply elements of that sequence to the split-cycle oscillator. Intrinsic evolution should provide those answers.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant Numbers CNS-1239196, CNS-1239171, and CNS-1239229.

References

[1] R. J. Wood, "The first takeoff of a biologically inspired at-scale robotic insect," *Robotics, IEEE Transactions on*, vol. 24, no. 2, pp. 341–347, 2008.

[2] Z. Teoh, S. Fuller, P. Chirarattananon, N. Prez-Arancibia, J. Greenberg, and R. Wood, "A hovering flapping-wing microrobot with altitude control and passive upright stability," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, Oct 2012, pp. 3209–3216.

[3] D. Doman, M. Oppenheimer, and D. Sigthorsson, "Dynamics and control of a minimally actuated biomimetic vehicle: Part I - aerodynamic model," *AIAA Guidance Navigation Control Conference*, 2009.

[4] ——, "Dynamics and control of a minimally actuated biomimetic vehicle: Part II - control," *AIAA Guidance Navigation Control Conference*, 2009.

[5] B. M. Perseghetti, J. A. Roll, and J. C. Gallagher, "Design constraints of a minimally actuated four bar linkage flapping-wing micro air vehicle," in *Robot Intelligence Technology and Applications 2*. Springer, 2014, pp. 545–555.

[6] S. K. Boddhu, H. V. Botha, B. M. Perseghetti, and J. C. Gallagher, "Improved control system for analyzing and validating motion controllers for flapping wing vehicles," in *Robot Intelligence Technology and Applications 2*. Springer, 2014, pp. 557–567.

[7] H. V. Botha, S. K. Boddhu, H. B. McCurdy, J. C. Gallagher, E. T. Matson, and Y. Kim, "A research platform for flapping wing micro air vehicle control study," in *Robot Intelligence Technology and Applications 3*. Springer, 2015, pp. 135–150.

[8] J. Gallagher, D. Doman, and M. Oppenheimer, "The technology of the gaps: An evolvable hardware synthesized oscillator for the control of a flapping-wing micro air vehicle," *Evolutionary Computation, IEEE Transactions on*, vol. 16, no. 6, pp. 753–768, Dec 2012.

[9] J. C. Gallagher and M. W. Oppenheimer, "An improved evolvable oscillator and basis function set for control of an insect-scale flapping-wing micro air vehicle," *Journal of Computer Science and Technology*, vol. 27, no. 5, pp. 966–978, 2012.

[10] J. C. Gallagher, L. R. Humphrey, and E. Matson, "Maintaining model consistency during in-flight adaptation in a flapping-wing micro air vehicle," in *Robot Intelligence Technology and Applications 2*. Springer, 2014, pp. 517–530.

[11] G. Greenwood and A. M. Tyrrell, *Introduction to Evolvable Hardware: A Practical Guide for Designing Self-Adaptive Systems*. Wiley-IEEE Press, 2006.

[12] M. North, N. Collier, and J. Vos, "Experiences creating three implementations of the REPAST agent modeling toolkit," *ACM Trans. Model. Comput. Simul.*, vol. 16, no. 1, pp. 1–25, 2006.

[13] R. Brooks, "A robust layered control system for a mobile robot," *Robotics and Automation, IEEE Journal of*, vol. 2, no. 1, pp. 14–23, Mar 1986.

[14] L. Sean, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Ballan, "MASON: A multi-agent simulation environment," *Simulation: Transactions of the society for Modeling and Simulation International*, vol. 7, no. 82, pp. 517–527, 2005.