Polymorphic Control Reconfiguration in an Autonomous UAV with UGV Collaboration

Corey Ippolito
Adaptive Control and Evolvable Systems Group
NASA Ames Research Center
Moffett Field, CA 94035
650-604-1605
corey.a.ippolito@nasa.gov

Sungmoon Joo
Department of Aeronautics and Astronautics
Stanford University
Stanford, CA, 94305
650-387-4715
joosm@ Stanford.edu

Khalid Al-Ali, Yoo Hsiu Yeh
Carnegie Mellon Innovations Lab
Carnegie Mellon University West Cost Campus
Moffet Field, CA 94035
650-861-6000, 361-655-6873
alali@cmu.edu, yoohsiu.yeh@west.cmu.edu

Abstract—The emergence of distributed technologies as a reliable infrastructure for real-time control is enabling a new generation of distributed plug-and-play control architectures and methodologies; increasingly common are control systems that pass real-time data across traditional system boundaries to utilize distributed remote sensing, processing, and actuation. The Polymorphic Control Systems (PCS) project formalizes constructs that permits topological reconfiguration of control systems that span multiple heterogeneous systems and multiple communication mediums, towards the goal of control coordination and strategy optimization in a multi-system environment, increased resilience to failure and uncertainty, increased overall and individual performance, and better utilization of available resources. This paper presents the concepts behind PCS, and presents results from a flight test experiment involving distributed reconfiguration of an autonomous landing controller in a collaborative multi-vehicle environment. These flight test experiments demonstrate one of the goals of polymorphic reconfiguration: providing emergency assistance and collaborative coordination between multiple systems to achieve safely the mission critical objectives, where a system failure would have resulted in the loss of the aircraft.[1,2]

TABLE OF CONTENTS

1. INTRODUCTION

The continuing maturation of distributed wireless technologies is evident in the growing proliferation of active research which is springing from these areas, such as distributed sensor networks, control systems, and distributed plug-and-play avionics infrastructures [1]-[6]. Much of this research is spurred by the proliferation of low-cost secure wireless communication hardware, such as wireless Ethernet hardware and next generation wireless cellular technology, resulting in the emerging ubiquity of wireless technologies in our everyday lives and in a growing number of domains. In particular, these recent advances are enabling new methods and techniques of control reconfiguration utilizing remote avionics, actuation, and sensing. Distributed plug-and-play concepts can be applied [1][4] to establish dynamic distributed avionics networks as the backbone for communications in a single vehicle or across multiple vehicle systems with the goal of enhanced performance, resilience, and fault-tolerance. These dynamic

1——————————
[1] IEEEAC paper #1645, Version 8, Updated December 13, 2007
[2] U.S. Government work not protected by U.S. copyright.

networks have been shown to allow for the instantaneous restructuring of coordinated control systems topologies in a group of vehicles that could include delocalized sensor, actuator, and controller components to establish highly unusual control configurations, providing fault-tolerance to a wider class of vehicle system failures that previous approaches were ill-equipped to handle. Control systems in these dynamic networks could conceivably be capable of instantaneous *polymorphic* change - that is, the instantaneous and fundamental restructuring of the controller form and function. Polymorphic control architectures could provide on the fly reconfiguration to optimize a controller topology given radical changes in the environment. The Polymorphic Control Systems project seeks to research and formulate concepts for analysis and synthesis of component-based control systems towards the realization of polymorphic control concepts, with the ultimate goal of increased performance, resilience, and fault-tolerance.

This paper presents results from PCS inspired flight test experiments conducted in 2007 at NASA Ames Research Center in Moffett Field, California. These experiments involved coordinated control between ground vehicle assets at a landing site and an autonomous unmanned aerial vehicle experiencing a mid-flight emergency landing, but without sufficient onboard sensing to conduct a safe landing. The lack of sufficient observability through its onboard systems forces the aircraft to consider wireless communication medium for closed loop control, utilizing the in situ resources of the airfield – in this case, an autonomous ground vehicle with a sensor suite designed for vision-based navigation of the UGV. Through polymorphic restructuring of both the onboard flight system and the ground vehicle system, and utilizing secure communication over wireless 900Mhz ISM-band, the ground rovers provide vision-based guidance and sensing to the aircraft.

2. POLYMORPHIC CONTROL SYSTEMS

The Polymorphic Control Systems approach applies control theoretic analysis and synthesis techniques to a mathematical construct that describes the composition and function of a distributed component-based system conducive to vehicle control system formulation and implementation, and implements these strategies on an embedded system architecture. The requirements for this approach can be conceptually divided into three main constituents: a *physical layer*, a *topological construct*, and an *analysis/synthesis approach*.

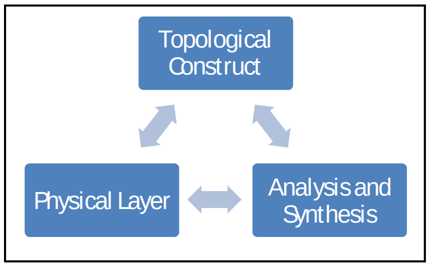


Figure 1. Conceptual Components of PCS

The ***physical layer*** refers to the actual implementation of the distributed embedded plug-and-play environment that allows for immediate on-the-fly reconfiguration over local and global vehicle systems. The physical layer comprises a number of technologies and mediums, including protocols for communication, the physical communication buses (e.g., Spacewire, MIL-STD-1553, etc.), various computing hardware communicating over the buses, wireless communication transceivers and radios, and the actual implementation of the algorithms that result from the PCS design process.

The PCS research utilizes the Reflection Architecture [8] - a plug-and-play middle-ware communications layer for real-time embedded systems - as the main protocol for inter-component communication. Reflection provides a large set of capabilities and functionality for component-based plug-and-play that meets the requirements of the physical layer PCS definitions [1]. The benefits of component-based approaches to development of large scale systems [9][10], and vehicle systems in particular [11], are well documented (see treatment in [1]). Reflection operates over a number of mediums, including the wireless mediums (900MHz, 802.11x) and wired buses (RS-232/422, Ethernet) used for these flight test experiments.

The ***topological construct*** defines the PCS component model: a mathematical description of a plug-and-play architecture as conceived for the purposes of PCS system formulation, analysis, and implementation. This layer defines a topological construct capable of describing a large class of control configurations, which must also allow for modeling and analysis of real-world properties of distributed computing architectures such as latency, bandwidth limitations, errors and uncertainty in the data signals. The PCS formulation provides concise definitions for a component-based plug-and-play system focusing on structural/topological definition and operations. The constructs model component-based intercommunications largely through a signal-routing architecture model, which is compatible with the implementation of Reflection in the physical layer.

The ***analysis/synthesis approach*** defines the analytical and synthesis engines required to solve PCS problems posed in the context of the topological construct, which in turn generates control algorithms and procedures to implement these results. Subsequently, the algorithms are implemented on the physical layer.

2.1 Modeling Systems in PCS

A component is a distinct, composable transform block whose interface is defined by its input and output characteristics, and whose implementation is strictly encapsulated. As an example, consider a linear time-invariant (LTI) dynamic system whose evolution is described by the vector-valued relationships shown in Figure 2. The vector **x** is composed of a set of state parameters, the set (**A**,**B**,**C**,**D**) are matrices that represent static properties of the component, both of which are encapsulated by the component in the PCS definition along with the system of equations (although the parameters can be exposed in the interface at the discretion of the developer). The set of input variables and output variables, **u** and **y** respectively, represent the interface of the component (in the example, shown as $\mathbf{u}\in\Re^n,\mathbf{y}\in\Re^m$).

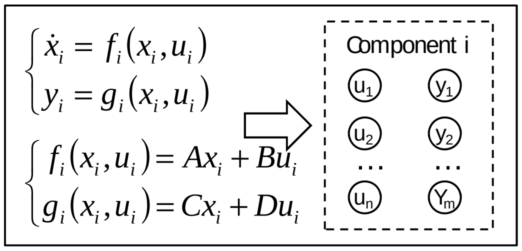

$\begin{cases}\dot{x}_i = f_i(x_i,u_i)\\ y_i = g_i(x_i,u_i)\end{cases}$
$\begin{cases}f_i(x_i,u_i)=Ax_i+Bu_i\\ g_i(x_i,u_i)=Cx_i+Du_i\end{cases}$
Component i
u_1
y_1
u_2
y_2
…
u_n
Y_m

Figure 2. LTI System Represented in PCS.

Complex dynamic systems are often decomposed into smaller functional blocks that can be characterized by their input and output relationships for the sake of interconnection and dataflow analysis. For instance, the linear system shown in Figure 2 may be decomposed as follows:

$$\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\end{bmatrix}=\begin{bmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix}+\begin{bmatrix}B_{11} & B_{12}\\ B_{21} & B_{22}\end{bmatrix}\begin{bmatrix}u_1\\ u_2\end{bmatrix}$$

$$y = C_{11}x_1 + D_{11}u_1 \qquad (1)$$

By defining vectors z_1 and z_2, the equation in (1) can be expanded to two distinct interconnected systems, such as:

$$\begin{cases}\dot{x}_1 = A_{11}x_1 + B_{11}u_1 + z_2\\ z_1 = A_{21}x_1 + B_{21}u_1\\ y = C_{11}x_1 + D_{11}u_1\end{cases}$$

$$\begin{cases}\dot{x}_2 = A_{11}x_2 + B_{22}u_2 + z_1\\ z_2 = A_{12}x_2 + B_{12}u_2\end{cases} \qquad (2a,b)$$

This componentized decomposition is represented in graphical form that specifies the input/output characteristics of the component, shown in Figure 3.

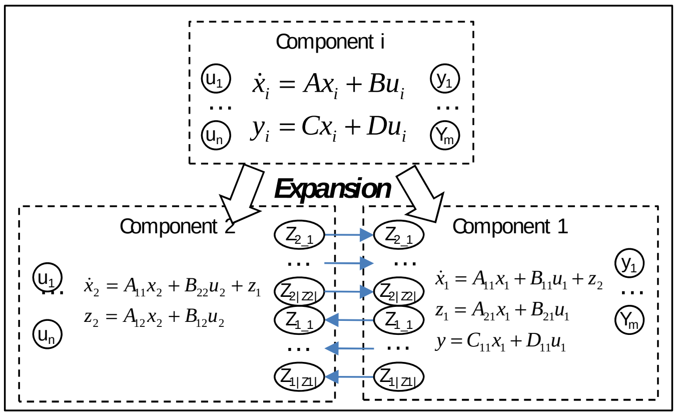

Component i
u_1
$\dot{x}_i = Ax_i + Bu_i$
$y_i = Cx_i + Du_i$
y_1
u_n
Y_m
Expansion
Component 2
u_1
u_n
$\dot{x}_2 = A_{11}x_2 + B_{22}u_2 + z_1$
$z_2 = A_{12}x_2 + B_{12}u_2$
Z_{2_1}
$Z_{2|Z2|}$
Z_{1_1}
$Z_{1|Z1|}$
Component 1
$\dot{x}_1 = A_{11}x_1 + B_{11}u_1 + z_2$
$z_1 = A_{21}x_1 + B_{21}u_1$
$y = C_{11}x_1 + D_{11}u_1$
y_1
Y_m

Figure 3. Expansion of an LTI System Component.

Similarly, consider a system of n components defined either by the general dynamic system formulation, or the LTI system formulation, as shown in Figure 2. These systems of interconnected components can be contracted into a single formulation of interconnected systems, for instance through an evolved system approach [7] as shown in eq 3. Below. Here, ε_{ij} is a parameter which controls the evolution of the component systems.

$$system\begin{cases}\dot{x} = f(x,u)\\ y = g(x,u)\end{cases}$$

$$\dot{x}_i = f_i(x_i,u_i) + \sum_{j=1}^{L}\varepsilon_{ij}g_{ij}(x_j,u_j) \qquad (3)$$

$$where \quad x = [x_1 \ldots x_L]^T \;, \; 0 \le \varepsilon_{ij} \le 1$$

In a topological context, the graph for this contraction is shown in Figure 4.

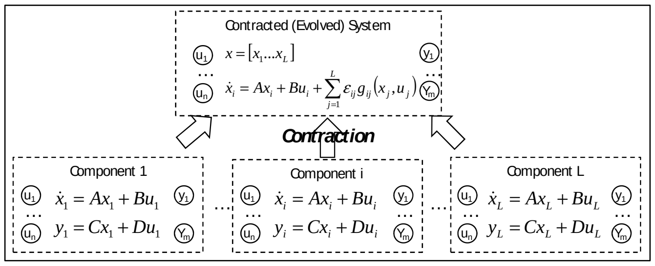

Contracted (Evolved) System
$x = [x_1 \ldots x_L]$
$\dot{x}_i = Ax_i + Bu_i + \sum_{j=1}^{L}\varepsilon_{ij}g_{ij}(x_j,u_j)$
Contraction
Component 1
$\dot{x}_1 = Ax_1 + Bu_1$
$y_1 = Cx_1 + Du_1$
Component i
$\dot{x}_i = Ax_i + Bu_i$
$y_i = Cx_i + Du_i$
Component L
$\dot{x}_L = Ax_L + Bu_L$
$y_L = Cx_L + Du_L$

Figure 4. Contraction through Evolved Transformation

The process shown in Figure 3 and Figure 4 are examples of more general topological operations defined as component ***expansion*** and component ***contraction***. The directed edges in these figures represent data flow and equivalence; that is, two variables are equivalent, and the directionality of the edge represents movements from the output of one component to the input of another component. The graphs of Figure 3 and Figure 4 shows *configuration space* representations of the same system from a functional point of view. Through the non-isomorphic operations of expansion and contraction, many different representations can be posed of the same system from a functional and dynamic standpoint.

2.2 Topological Construct

Conceptually, the following PCS construct provides a formalization describing control systems from a graph-theoretic topological standpoint, which is uncommon in the literature for real-time control problems. The basic constructs of the PCS framework are similar to graph theoretic definitions that formalize hypergraphs, but have distinct and important differences. Let a *vertex* v∈V be an indivisible unit in the component graph. Vertices represent a data attribute of a component in a component graph. Let a *directed edge* e∈E be defined as an ordered pair e=<s,t> where s,t∈V. Edges represent a route for data flow between two data attributes. Let E be the set of all edges. For notation purposes, let the mappings *init*:E→V and *ter*:E→V be defined on every edge e∈E such that *init*(e)=s and *ter*(e)=t, where s∈V is the initial vertex and t∈V is the terminal vertex of the edge, respectively.

A *component graph* G=(V,E,C) is a set of vertices V, a set of edges E, and set of components C that is recursively defined below. Let a graph G'=(V',E',C') be a subgraph of a graph G=(V,E,C) if V'⊆V, E'⊆E, and C'⊆C. "G is a subgraph of G'" is written as "G⊆G'".

A component graph G is *edge contained* if the following holds: *init*(e)∈*V*(K) and *ter*(e)∈*V*(K) ∀ e∈*E*(K).

A *component* K∈*C*(G) is a component graph subject to the following constraints:

(i) K is a subgraph of G;

(ii) *init*(e)∈*V*(K) and *ter*(e)∈*V*(K) ∀ e∈*E*(K) (Edge Containment);

(iii) if L∈*C*(K) then L is a subgraph of K (Component Containment), and

(iv) K∉*C*(K) (Monotonicity).

For a component K, an element of *C*(K) is a called a subcomponent of K. All components must have at least one vertex. If a component contains an edge, it must start and end in vertices contained within that component. All subcomponents of a component are proper subgraphs of that component, and a component cannot contain itself. As a result, subcomponents are a proper subset of their owning component. A component C *subsumes* a component K iff K⊂C and either (i) c'∈C(c), or (ii) there is a subcomponent c''∈C(c) that subsumes c'. Note that a component can never subsume itself.

The definition of a component as a subgraph allows an arbitrary level of detail when describing components in the system, as shown in Figure 5. Non-trivial systems can be described as a single component that contains the entire component graph G.

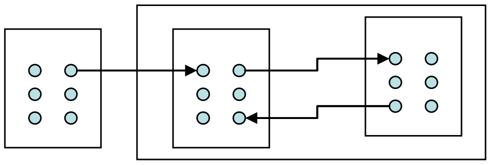


Figure 5. Component Level of Detail

The configuration space defines the space of all possible configurations, i.e., the space of all graphs representing all possible combinations of components and interconnections. Let the *configuration space* $\mathcal{C}$ be defined as the graph $\mathcal{C}$=(V,E,C) where

(i) V($\mathcal{C}$) is a set of vertices,

(ii) C($\mathcal{C}$) is a set of components, and

(iii) E($\mathcal{C}$) is a set of all possible directed edges on V; i.e. <s,t>∈E($\mathcal{C}$) ∀ s,t∈V(G).

For notation purposes, given a graph G=(V,E,C), let the mappings *V*(G)=V, *E*(G)=E, and *C*(G)=C be defined for every graph G. In addition, an entity x∈G is equivalent to x∈*V*(G)∪*E*(G)∪*C*(G). A component C contains a set of vertices V if V⊆*V*(C).

Let the *parent component* of a vertex v be defined as follows: for every vertex v∈V in $\mathcal{C}$, there exists a unique c∈*C*($\mathcal{C}$) such that v∈c, *E*(c)=∅, and *C*(c)=∅. This unique component c is said to be the parent of v, and is written *parent*(v) = c.

A vertex has one and only one parent component. As a result of this rule, the set of parent components in C($\mathcal{C}$) are set of components with no edges or subcomponents, and this set is unique and disjoint. Further, there are components with no edges or subcomponents that are inadmissible in $\mathcal{C}$ because of violation of the parent component definition.

The concept of connectiveness from traditional graph theory is useful, and is derived here for components in a graph. Note that these definitions do not consider edge direction in the definitions for component-connectivity.

Given a graph G, two components A,B∈*C*(G) are *neighbors* in G if there exists an edge e∈*E*(G) such that (*init*(e)∈*V*(A) ∧ *ter*(e)∈*V*(B)) ∨ (*init*(e)∈*V*(B) ∧ *ter*(e)∈*V*(A)).

Let a *component-path* P_c be an ordered set of components in G where either (i) $P_c=\langle A,B\rangle \wedge$ A,B are neighbors, or (ii) there exists two components $A,B\in P_c$ such that A,B are neighbors, and the set (P_c-{A}) is a component-path. A component graph G is *component-connected* if there exists at least one component-path between every pair of components in G. A component-path $P=\langle X_0,X_1,\ldots,X_{n-1},X_0\rangle$ where $n\geq 2$ is a *component-cycle*.

A component-cycle P is of maximal length in G if for all $P'\subseteq C(G)$, where $P'\neq P$ and P' contains a component-cycle, $|P|\geq|P'|$. A component graph G is a *cycle-bound component graph* if two conditions hold: (i) G is component-connected, and (ii) there exists a path P* of maximal length in G where, given a component $K\in C(G)$, either (a) $K\in P^*$, or (b) there exists a component-cycle P' in G where $K\in P'$ and $P^*\cup P'\neq\varnothing$. In other words, every component is either part of the maximal length component-cycle, or is contained in another component-cycle which shares at least one component with the maximal length component-cycle.

Given these definitions we can formally define a particular configuration in configuration space, where a configuration is a graph that represents a finite selection set of components, edges and vertices. Let a *configuration graph* G in $\mathcal{C}$ be defined as a graph G=(V,E,C) that represents a specific configuration implementation, subject to the following:

(i) $G\in C(\mathcal{C})$ (G is a Component);

(ii) For all $v\in V(G)$ there exists one and only one $K\in C(G)$ such that $v\in K$ (Disjoint Subcomponents);

(iii) For all $e\in E(G)$, let $init(e)\in V(L)$ and $ter(e)\in V(M)$ are contained in different components (Pruned Edges);

(iv) G is component-connected (Component-Connectedness).

For a configuration graph, all components in the representation are disjoint; in other words vertices are contained by only one component in G, which is not the parent of the vertex if $parent(v\in G)\notin C(G)$.

The following statements (see [1]) provide some insight into necessary conditions for observability and controllability of components through topological properties of a controller graph.

Consider a component graph G. If a component $K\in C(G)$ is *controllable* in G, then there exists a path P in G where $K\in P$, and an edge e in P where $term(e)\in V(K)$.

Consider a component graph G. If a component $K\in C(G)$ is *observable* in G, then there exists a path P in G where $K\in P$, and an edge e in P where $init(e)\in V(K)$.

2.3 Physical Layer

The physical layer in the Polymorphic Control Systems formulation must implement a morphable controller network topology over continuous systems and components, as defined in the topological construct. In addition to supporting the component-based definitions, a candidate PCS physical layer must also support the following operations.

Let G be a component in $\mathcal{C}$ with pruned edges and disjoint subcomponents. Define the component-prune operation *cprune*(G) to be as follows. If there exists a component $A\in C(G)$ such that A is not component-connected to any other component $B\in(C(G)-A)$ for all B, then *cprune*(G)=(*V*(G),*E*(G),*C*(G)-A). Otherwise *cprune*(G)=G.

Let G and K be non-empty component graphs in $\mathcal{C}$, let $e\in E(\mathcal{C})$, $e\neq 0$, such that e connects G and K. Then define *combineCGraphs*(G, K, e)=(*V*(G)+*V*(K), *E*(G)+*E*(K)+e, *C*(G)+*C*(K)). The result of *combineCGraph*(G, K, e) is itself a component graph.

Let G be a configuration graph in $\mathcal{C}$, let $e\in E(\mathcal{C})$, $e\neq 0$, and $ter(e)\cup init(e)\subseteq V(G)$, then the operation *addEdge*(G,e) is defined by the following:

(i) If no component in G subsumes *ter*(e), then *addEdge*(G,e)=*combineCGraph*(G, *parent*(*ter*(e)), e);

(ii) Else if no component in G subsumes *init*(e), then *addEdge*(G,e)=*combineCGraph*(G, *parent* (*init*(e)), e);

(iii) Otherwise *addEdge*(G,e) = G.

Let the operator $\varnothing$ be defined as $G\varnothing e$:= *cprune*((*V*(G), *E*(G)-e, *C*(G)), where G is a configuration graph in $\mathcal{C}$, and $e\in E(\mathcal{C})$. Similary, let the operator $\oplus$ be defined as $G\oplus e$:= (*V*(G),*E*(G)+e,*C*(G)). Note that $G\oplus e$ and $G\varnothing e$ results in a configuration graph.

In implementation, these definitions are implemented through hardware or software algorithms that are specific to the particular layers involved.

3. Flight Test Experiments

PCS flight test experiments were conducted that focused on a specific case of reconfiguration where observability of the system has been damaged or degraded, and no possible reconfiguration onboard the vehicle system would provide the necessary conditions for a safe landing to occur. In this situation, traditional control strategies for failure mitigation - such as reconfiguration (recovering from actuator failure), robust design (uncertainty), or adaptive control strategies (actuator failure or uncertain dynamics) - would not be sufficient to recover and save the aircraft. The objective of the experiments was to demonstrate through flight-testing the viability of real-time polymorphic reconfiguration, demonstrate the PCS algorithms effectiveness in time-critical control applications, and demonstrate the proper operation of the real-time embedded software that is hosting the PCS algorithms.

Figure 6. Flight Testing at Moffett Field, CA

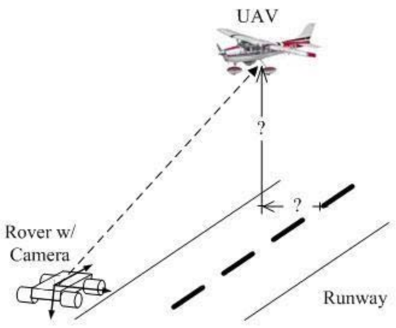


Figure 7. UGV Assist in UAV Landing

The experiment was designed as follows. A UAV performing flight maneuvers at altitude is required to perform an immediate landing maneuver. The UAV is assumed to have suffered damage to its onboard position estimation sensors, and accurate ground-relative position measurements - particularly AGL altitude, glide slope, and localizer deviation measurements - are not available to conduct a landing. A ground-based unmanned autonomous rover is in the nearby vicinity, monitoring the airfield for debris and foreign objects utilizing its onboard sensors suite that includes a vision-based navigation system to identify and track airfield debris hazards. The PCS system is utilized to model the system, analyze and reconfigure the controllers onboard both vehicles, and conduct the aircraft to a safe and timely landing.

The limited accuracy of low-cost position measurement sensors causes difficulties for autonomous landing of small-scale low-cost UAVs. Many small-scale UAV systems utilize low cost avionics suites with standard GPS, which provides position accuracy of approximately 10m (with 95% confidence) in the horizontal direction and 15m (with 95% confidence) in the vertical direction, which is insufficient accuracy to perform a flare and landing maneuver. Additionally, GPS does not provide ground-relative estimations, and landing is not possible without accurate measurement of the ground altitude on the runway. While these are real issues faced by UAV designers and developers, the purpose of this experiment is not to design a UAV landing system. Rather, these experiments demonstrate a scenario where, at a certain point in time, the global system is faced with a scenario where it cannot meet desired objectives (due to damage of onboard components, for instance), and state of the art control techniques will not be able to complete the mission objectives or even save the aircraft.

The PCS flight test experiments were conducted on Moffett Field (Figure 6) at NASA Ames Research Center on two research vehicle platforms: the Exploration Aerial Vehicle (EAV) UAV platform, and the Mobile Autonomous eXplorer (MAX) UGV platform.

3.1 EXPLORATION AERIAL VEHICLE UAV

The Exploration Aerial Vehicle (EAV) [12] is an unmanned autonomous aerial vehicle build on a Hanger 9 airframe modeled after the 2000 version of a Cessna 182 at one quarter-scale (Figure 8). This particular airframe affords a large interior volume for installing flight avionics and systems. The specifications for the EAV are shown in Table 1.

Figure 8. The Exploration Aerial Vehicle (EAV)

Table 1. Exploration Aerial Vehicle Specifications

Airframe	Hanger 9 Cessna 182 Skylane 95" ARF
Wing Span	94.75 in (2406 mm)
Overall Length	76.75 in (1949 mm)
Wing Area	1246 sq in (80.39 dm²)
Wing Loading	32.7 oz/sq ft
Flying Weight (Empty)	18.5 lb (8.22 kg)
Flying Weight (Full)	23.2 lb (10.52 kg)
Max Payload Weight	10 lbs
Cruise Speed	45 knots
Operations Ceiling	500 ft (flight field restrictions)
Engine Make/Model	Zenoah G-38
Engine Type	2-Stroke Gas/Oil
Engine Displacement	2.3 cu in (38 cc)
Actuation/Servomotors	Six (6) HiTec HS-5646MG DC Programmable Digital Ultra Torque Servos
Primary CPU	Diamond Athena 660MHz/128MB RAM
Secondary CPU	Versalogic Cheetah M 1.6/512MB RAM
Embedded Controller	Motorola DSP56807
Sensor Suite	Athena GS111m INS/GPS Unit, provides full 6DOF state, WAAS-enabled GPS, angle of attack, sideslip, airspeed and pressure altitude
Sensors/Vision	Point Grey Dragonfly Cameras
Communication Links	72Mhz Receiver (Pilot/ Safety Control), 900Mhz Transceiver (Data Communications), 2.4GHz Transceiver (Data/ Video Downlink)

A set of flight tests had been conducted previously to identify the major lateral and longitudinal modes of the EAV. During these tests, specific maneuvers (such as 3-2-1-1, 2-1-1, pulses, and doublets) applied to the aircraft excited the aircraft modes sufficiently for system identification. A least-squares regression in frequency domain identified the major modes of the system, as shown in Figure 9.

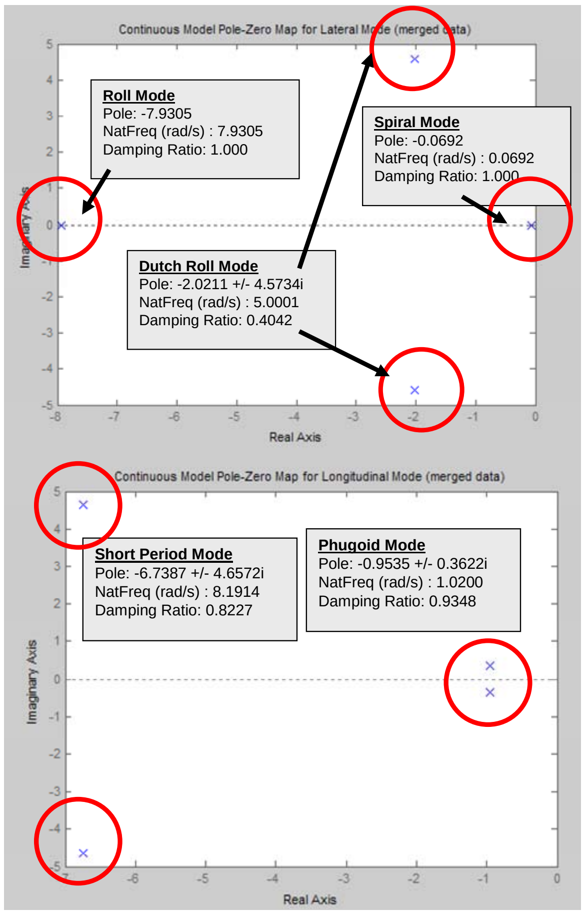


Figure 9. EAV System: Pole-Zero Plots of the Lateral and Longitudinal Modes

Table 2. EAV System Characteristics

Mode	Pole	Frequency (rad/s)	Damping (ξ)
Roll Mode	-7.9305	7.9305	1.0000
Dutch Roll Mode	-0.0692	5.0001	0.4042
Spiral Mode	-0.0692	0.0692	1.0000
Short Period Mode	-6.7387 +/- 4.65721i	8.1914	0.8227
Phugoid Mode	-0.9535 +/- 0.36220i	1.0200	0.9348

3.2 Mobile Autonomous eXplorer UGV

The Mobile Autonomous eXplorer (MAX) UGV platform is a commercial robotics platform designed by Carnegie Mellon University West Campus and Senseta, Incorporated (Figure 10). This UGV is a small all-terrain vehicle used for research and education, with a powerful and densely packed sensor and computing suite detailed in Table 3.

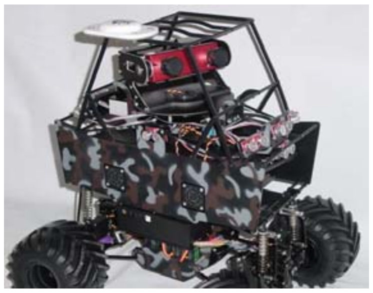


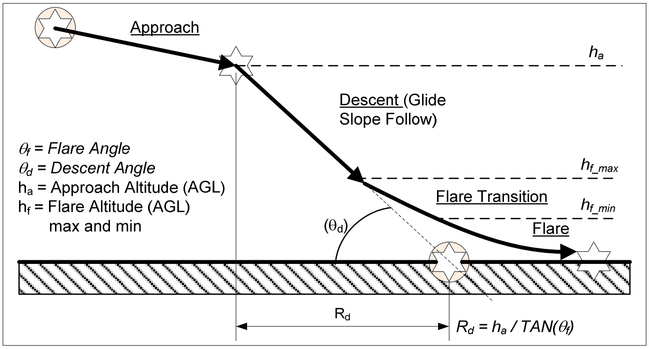
Figure 10. The Senseta Inc. Mobile Autonomous eXplorer (MAX) Ground Rover

Table 3. MAX Ground Rover Specifications

Airframe	Senseta, Inc. MAX Rover, version 5.0A (Ames), Carbon-Fiber Frame
Dimensions	18” x 15” x 19”
Top Speed	11.2 mph (5.0 m/s)
CPU – General Purpose	1.8 GHz Pentium-M, 1GB RAM Mini-ITX Board with Full-Size PCI Expansion Slot
CPU - Embedded	Two (2) Programmable Onboard I/O Boards based on the Motorola DSP56F807 Chipset
Drive Train/Suspension	Two (2) Novak SS4300 Brushless DC Motor Systems, Four wheel drive with front and rear differentials, Beam Suspension
Turning Radius	Twin, independent Ackerman steering with tight turning radius (25 cm)
Sensing- Inertial/GPS	Athena Guidestar GS-111m Navigation System (3-axis accelerometer, 3-axis gyroscope, 3-axis magnetometers) with DGPS (cm accurate)
Sensing- Sonars	10 sonar rangefinders with integrated photo sensors
Sensing- LIDAR	Six Hokuyo Scanning LIDAR, 240o Scan Angle, 1024 Lines at 50Hz, 4m range, +/-10mm
Sensing- Vision	Stereo camera pair (640x480 @ 30fps or 1280x960 @ 7.5fps 24-bit color) mounted on an articulated panospheric pan and tilt unit

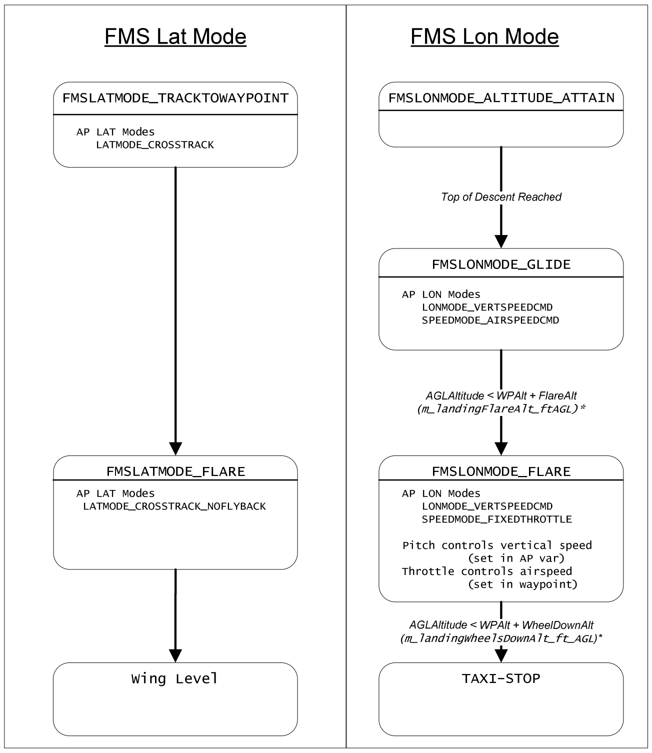
3.3 AUTOLANDING SYSTEM

The onboard flight system and ground rover system were designed and implemented in the Reflection Architecture, allowing PCS reconfiguration to occur on the fly in this flight test experiment on the EAV flight computer, just prior to the approach phase. The vision processing components were implemented using Matlab Simulink on the ground rovers; unfortunately, time constraints did not permit the vision processing component to be ported over to Reflection, so the ground rover system was manually configured to the final PCS control graph configuration, and did not reconfigure on the fly.



Approach
h_a
Descent (Glide Slope Follow)
θ_f = Flare Angle
θ_d = Descent Angle
h_a = Approach Altitude (AGL)
h_f = Flare Altitude (AGL) max and min
h_{f_max}
Flare Transition
h_{f_min}
Flare
(θ_d)
R_d
$R_d = h_a / TAN(\theta_f)$

Figure 11. Aircraft Landing Profile



FMS Lat Mode
FMSLATMODE_TRACKTOWAYPOINT
AP LAT Modes
LATMODE_CROSSTRACK
FMSLATMODE_FLARE
AP LAT Modes
LATMODE_CROSSTRACK_NOFLYBACK
Wing Level
FMS Lon Mode
FMSLONMODE_ALTITUDE_ATTAIN
Top of Descent Reached
FMSLONMODE_GLIDE
AP LON Modes
LONMODE_VERTSPEEDCMD
SPEEDMODE_AIRSPEEDCMD
AGLAltitude < WPAlt + FlareAlt
(m_landingFlareAlt_ftAGL)*
FMSLONMODE_FLARE
AP LON Modes
LONMODE_VERTSPEEDCMD
SPEEDMODE_FIXEDTHROTTLE
Pitch controls vertical speed (set in AP var)
Throttle controls airspeed (set in waypoint)
AGLAltitude < WPAlt + WheelDownAlt
(m_landingWheelsDownAlt_ft_AGL)*
TAXI-STOP

Figure 12. FMS Autopilot Landing Logic and Mode Transition Diagram

Conceptually, the EAV controller is composed of three major components: the flight management system (FMS), the mode-based autopilot system, and the hardware/actuator interfaces (see Figure 13 for details of a landing mode configuration). The FMS onboard the UAV is responsible for monitoring the aircraft, receiving instructions from the ground station computers, managing the flight logic, and instigating the appropriate mode transitions of the lower-level autopilot systems. This responsibility includes implementing the autopilot landing system logic.

The autopilot landing system follows the profile shown in Figure 11. Figure 12 shows the FMS state transition diagram for the autopilot landing system. This system has four phases: approach, descent/glide slope follow, flare

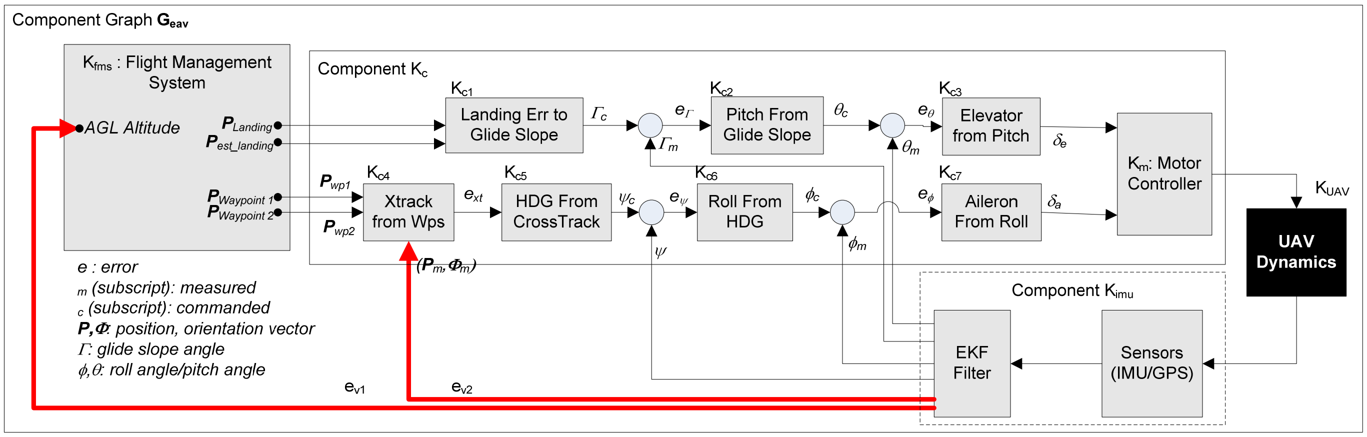


Figure 14. PCS Configuration Graph G_{ea}, for the EAV Glide Slope Phase Autopilot
The highlighted signals have accuracies which are insufficient for a safe landing.

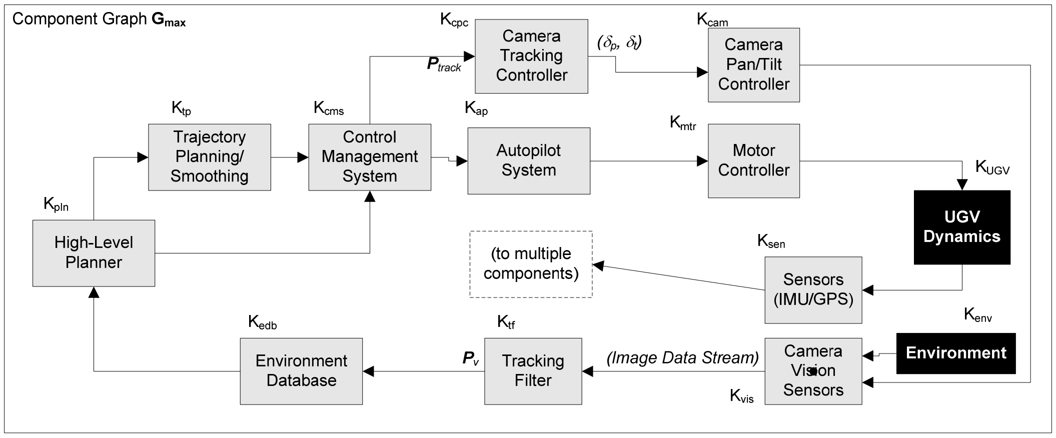


Figure 14. PCS Configuration Graph for the Conceptual UGV Autonomous Control System

transition, and flare. In the approach phase, the aircraft slows to the reference approach speed and descends from cruising altitude to the descent altitude. The controller then follows a descent trajectory by following a controlled glide slope angle that varies between 0.0 and 10.0 degrees (5 degrees nominal) based on the error between the desired landing point and the aircraft's estimated landing point. At a predefined transition altitude, the aircraft transitions from the glide-slope controller to the flare controller. During the autopilot testing, the UAV system was found to provide more consistent performance with the introduction of a flare transition phase; this transition phase gradually transitions from the glide-slope follow to the flare trajectory. Once the UAV enters the final flare phase, the aircraft throttles to a minimum setting, with a fixed descent rate until touchdown.

3.4 PCS ANALYSIS

Figure 13 shows a PCS configuration graph for one of the autopilot modes; in this case, the 'descent' phase of the autopilot landing system. In this configuration graph, the highlighted position signals have insufficient accuracies to achieve an autonomous landing. The component graph G_{eav} comprises the set of components $C(G_{eav})=\{K_{fms}, K_c, K_m, K_{UAV}\}$, and the component K_c can be further decomposed into $C(K_c)=\{K_{c1}..K_{c7}\}$. Here, K_{ci} are PID controller blocks. The UAV dynamics represented by the component K_{uav} represent the physical system rather than an implemented component. The inclusion of K_{uav} endows G_{eav} with several appealing properties: G_{eav} is an *edge contained*, *component connected*, *cycle-bound component-graph*. The removal of the edge set $E_v=(e_{v1},e_{v2})$ leaves G_{eav} without a maximal length component-cycle that would meet the requirements for G_{eav} to be cycle-bound, which is a necessary condition in the outer loop implementation for controllability of K_{fms}.

The ground vehicle's autonomous control system is shown in Figure 14 as a highly contracted high level PCS configuration graph, G_{max}. Similar to G_{eav}, the K_{UGV} and K_{env} are included for analysis, although G_{max} is not a maximal-cycle component graph (K_{env}, for instance, does not meet the necessary conditions for controllability). A path of interest in this graph is the path $\{K_{cpc}, K_{cam}, K_{vis}, K_{tf}\}$. K_{vis} is the vision processing system, which is an interface to the camera hardware and provides a steady stream of image data. K_{tf} is a vision processing/object detection filter which produces the position of a tracked object of interest. K_{cpc} is the camera pan/tilt head tracking controller, which points to an input position. Establishing a path from K_{tf} to K_{cpc} would form a closed loop tracking control around the camera system, aiming the camera at the current object of interest being identified by the tracking filter K_{tf}.

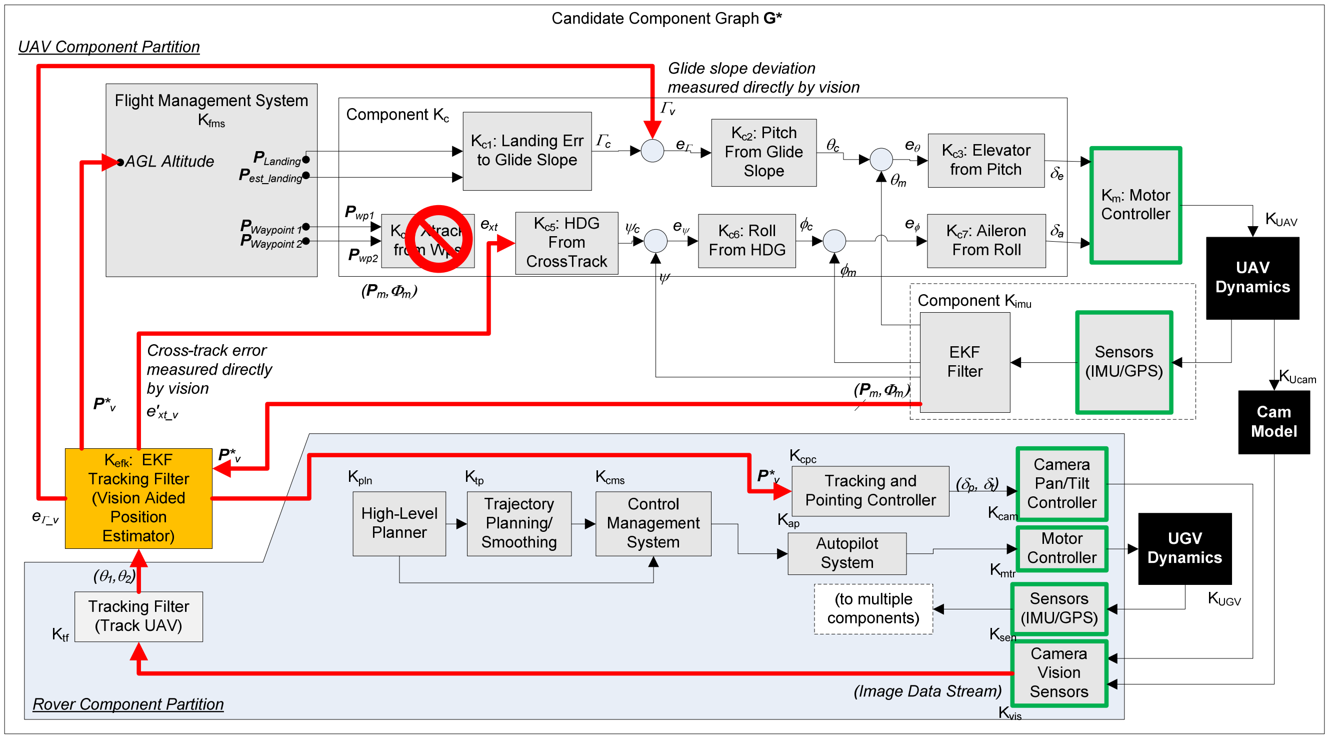


Figure 15. Configuration Graph G*

The highlighted signals and K_{ekf} were added.

Bold-outline blocks must remain in a particular system partition.

Problem Statement: The PCS problem statement can be stated in two parts. Given the graph defined by G'= *combineCGraph*(($G_{eav}\varnothing E_v$), G_{max}), find a sequence of operations on the graph G' that arrives at a graph $G^* \in \mathcal{C}$, where G* provides a control path that includes the components {K_{fms}, K_c, K_{uav}}, and G* stabilizes and controls K_{uav}. Given a candidate graph G*, determine the control strategy that provides guidance for the aircraft to safely conduct an autonomous landing.

Control Topology: Several candidate graph topologies can be constructed utilizing K_{tf} to track the UAV, but since no criteria for optimization or performance was enforced in this problem statement, the candidate graph G* was selected for ease of implementation, and is shown in Figure 15. In more general situations where additional candidate topologies can be considered, or multiple resources require consideration to constraints such as bandwidth or processing limitations, the graph theoretic constructs were designed to allow analysis and implementation of topological optimization operations, as described in [1].

The sequence of operations that operate on G_{uav} are shown in Figure 16. This solution required the development of a new controller component K_{ekf} that was not an element of the original graphs G_{eav} or G_{max}. This component contains a custom extended Kalman filter, designed to take the output angle provided by the image processing component K_{tf} and provide an estimate for the position of the aircraft. Note that since the edge from K_{tf} to K_{ekf} crosses the system boundary, latency will be incurred on the signal, and this is taken into account in the filter design. The details of the development of this filter are given in [2].

Once the new component K_{ekf} was added, a minimum-cut was determined to partition the components to the various partitions to minimize communication bandwidth usage. During this process, the K_{ekf} component was moved to the UAV system partition.

The graph G* is edge contained, component connected, and is a maximal-cycle component-graph (with the elimination of K_{c4}). The necessary conditions for observability and controllability on K_{fms} is provided by introducing the edge from K_{ekf} to K_{fms}, which transports the estimated position of the UAV based on vision (P^*_v).

Rover Control Strategy: Similar to the selection of the configuration topology G*, the control strategies on the control graph G* for both the rover and the UAV system were selected for sufficiency and ease of implementation. As shown in Figure 7, the rover is given a high level command to navigate to a position on the runway ahead of the landing zone. This requires manipulation of the planner (K_{pln}), but the control structure cycle {K_{pln}, K_{tp}, K_{cms}, K_{ap}, K_{mtr}, K_{ugv}, K_{sen}} was not modified from the original G_{max} configuration (edge K_{sen}-K_{pln} is not shown). From this vantage point, the rover can directly measure cross-track error, glide slope deviation, and can infer AGL altitude and position. The control path {K_{cpc}, K_{cam}, K_{vis}, K_{tf}} does exist as a cycle in G*, representing a closed loop control of the

```
// Reconfiguration operations

define Kekf;
define objMatlabIntrfc;

// Load Matlab Interface
objMatlabIntrfc = Guav.CreateComponent ( "pcsmatlabinterface.dll" );
        // ... objMatlabIntrfc specific function calls omitted...

// Delete and add components.
Guav.DeleteComponent ( Kc1 );
Guav.DeleteComponent ( Kc2 );
Guav.PruneEdges ();
Kekf = Guav.CreateComponent ( "pcstrackingfilter.dll" );

// Connect the graph
Guav.CreateEdge ( "Kimu.m_posEast_ft", "Kekf.m_posEast_ft" );
Guav.CreateEdge ( "Kimu.m_posNorth_ft", "Kekf.m_posNorth_ft" );
Guav.CreateEdge ( "Kimu.m_posUp_ft", "Kekf.m_alt_agl_ft" );
Guav.CreateEdge ( "Kimu.m_velEast_fps", "Kekf.m_velEast_fps" );
Guav.CreateEdge ( "Kimu.m_velNorth_fps", "Kekf.m_velNorth_fps" );
Guav.CreateEdge ( "Kimu.m_velUp_fps", "Kekf.m_velUp_fps" );
Guav.CreateEdge ( "Kekf.m_posNorthOut_ft", "Kfms.airplane_pos_north_ft" );
Guav.CreateEdge ( "Kekf.m_posEastOut_ft", "Kfms.airplane_pos_east_ft" );
Guav.CreateEdge ( "Kekf.m_alt_aglOut_ft", "Kfms.airplane_pos_altitude_ft" );
Guav.CreateEdge ( "Kekf.m_crossTrackAngleErr_rad", "Kc5.inputXTrackAngularErr_rad" );
Guav.CreateEdge ( "Kekf.m_crossTrackAngleErr_rad", "Kc2.inputGSAngleActual_rad" );
Guav.CreateEdge ( "objMatlabIntrfc.m_azimuthAngle_rad",
Guav.CreateEdge ( "objMatlabIntrfc.m_isRxExperimentData", "Kekf.m_isRxExperimentData" );
Guav.CreateEdge ( "objMatlabIntrfc.m_isVisionDataValid", "Kekf.m_isVisionDataValid" );
Guav.CreateEdge ( "objMatlabIntrfc.m_imageTime_sec", "Kekf.m_imageTime_sec" );
Guav.CreateEdge ( "objMatlabIntrfc.m_vAngle_rad", "Kekf.m_vAngle_rad" );
Guav.CreateEdge ( "objMatlabIntrfc.m_hAngle_rad", "Kekf.m_hAngle_rad" );
```

Figure 16. UAV Reconfiguration Script for G* (ReflectionScript Language)

tracking head. The tracking filter must be set to track the UAV, and the closed loop controller for the tracking head was included that will track the aircraft as the aircraft descends.

UAV Control Strategy: The manipulation to the UAV control system is largely topological.

In order to establish a controller that satisfies the objectives, the onboard UAV system reconfigures from G' to G*. One aspect of the reconfiguration is that data from the rover's remote sensors are routed to several points in the mid-level control loops. The reconfiguration bypasses the cross-track error calculation component K_{c4} completely, and feeds the cross-track error directly into K_{c5} from the vision measurement in K_{ekf}, in an attempt to close the loop around the remote sensors at the lowest level possible. Likewise, the reconfiguration routes the glide-slope measurement directly to the K_{c2} component.

Implementation Concessions for Flight Test: Unfortunately, the constraints of flight testing on a live runway and on schedule required a large number of concessions. The rover was positioned on the side of the runway, rather than a location in the center of the runway, because of flight safety issues. Additionally, the vision controller development required more time than was expected, which did not allow time for the Matlab controller to be ported to Reflection. As a result, the vision processing on the rovers had to be performed in Matlab at run-time, and the rover system could not take part in the reconfiguration; rather, the rover's configuration graph in G* was implemented completely in Matlab [2]; the rover position and camera articulations were fixed. Vision sensor data was directly uploaded to the UAV's system through a wireless 900Mhz radio modem link.

4.0 Flight Test Results

The flight control algorithms for PCS were successfully implemented and tested on the EAV UAV and the MAV UGV vehicle systems in a series of tests in the later part of 2007. The graphs in Figure 17 through Figure 20 at the end of this report show profiles and trajectories for many of the approaches.

The Reflection Architecture [8] was used for constructing and maintaining the PCS graphs. The algorithms onboard the EAV were implemented in C++ on a 700Mhz Pentium III class PC/104 processor. The reconfiguration script was written in ReflectionScript, and interpreted averaged 10.3 milliseconds to execute the reconfiguration on this platform. Several successful flight tests were conducted that tested the mid-flight control reconfiguration. The reconfiguration onboard occurred without any noticeable problems.

The ground rover systems were not implemented in a PCS-enabled architecture because of schedule constraints, but rather the final configuration was implemented in Matlab utilizing the image processing toolbox. The vision processing loop, running in Matlab on the 1.8 GHz mini-ITX CPU, ran at roughly 5-10 Hz, but varied depending on the image complexity. Development of the vision processing algorithms, including noise rejection, is covered in [2].

The amount of time required to develop the flight test was much longer than expected. In fact, the majority of the development and flight test time was spent tuning and modifying the landing controller, developing successful strategies for the rover vision systems [2], and interfacing with Matlab. These issues resulted in accomplishing only a fraction of the initial goals. The vision processing and EKF filter performed very well during the tests. The onboard EAV systems are capable of providing around 3.5-7m accuracy in altitude AGL. Through reconfiguration, the vision-based system provided less than 1.5m error. See reference [3] for further details.

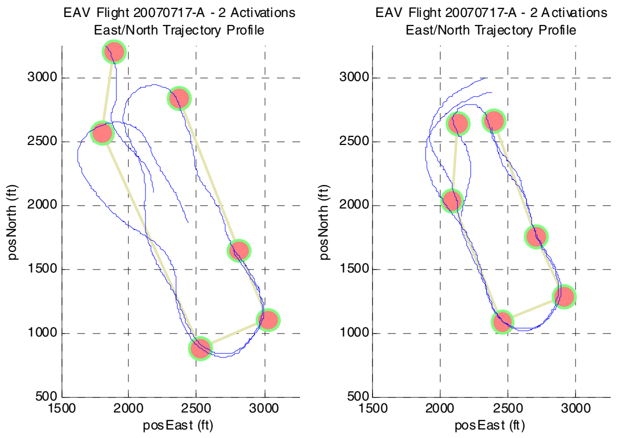


Figure 17. Ground-Track of Autonomous Landing.

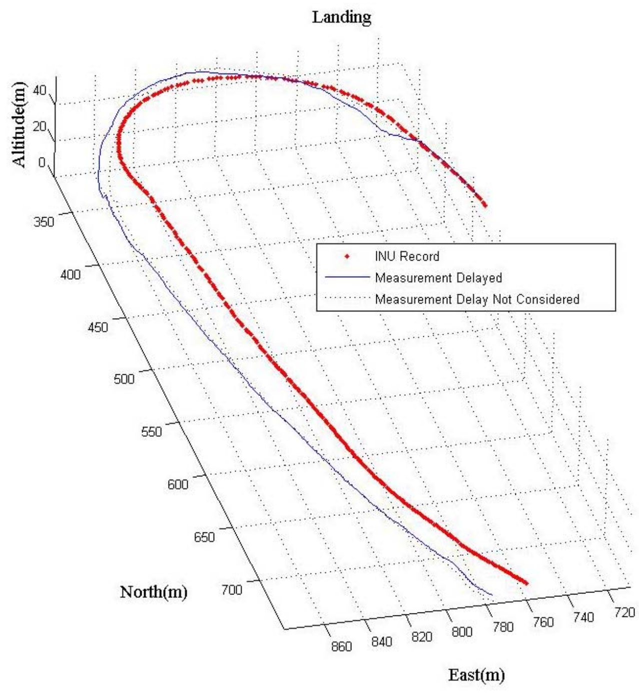


Figure 18 3D Plots of the Autonomous Landings.
Showing measurement delay consideration, see ref. [3])

4.0 Conclusion

The successful flight test of this small-scale PCS experiment fielded PCS inspired architectures and controllers for the first time on real flight vehicles. The goals for the PCS project are broad, and this simple and modest flight test experiment barely scratches the surface of possibilities for topological approaches to control reconfiguration over distributed networks. Previous experiments have shown the scalability of the PCS approach; PCS itself is in some sense a modeling tool, and using these tools scalable approaches have been introduced to automatically assemble controllers when competing multiple configuration possibilities over limited bandwidth and processing situations [1]. Both these previous applications and the experiment detailed in this paper show real-time configuration on various types of vehicle systems. These applications begin to shed light on how distributed wireless technology, lightweight plug-and-play architectures, and graph-theoretic topological analysis can be applied to the problem of control reconfiguration to develop a class of controllers that can adapt to certain environments, situations, and failures better than alternative state of the art techniques.

Further development into the mathematical formulation will be pursued. The conditions for stability, controllability, and observability are currently being investigated to see if graph theoretic approaches can provide new insight into control systems from a topological perspective. For instance, certain guarantees when contemplating controller/dynamic system composition and reconfiguration may be easier to achieve through topological analysis than through current approaches based on linear and non-linear analysis. This may lead to optimization methods for rejection or acceptance of candidate control topologies, control system optimizations in large distributed control systems from a control structures approach, or automated assembly of controllers in damaged systems through topology reconfiguration.

ACKNOWLEDGMENT

The authors wish to thank all those who have contributed their knowledge and expertise to make this research project a reality. Thanks to Kalmanje Krishnakumar, Mark Sumich, and the adaptive controls group at NASA Ames Research Center for their continued support of this project. Special thanks to student interns Mera Horne, Matt Wallach, Brad Jackson, Matt Walliser and Ace Shelander, whose hard work made these flight tests possible.

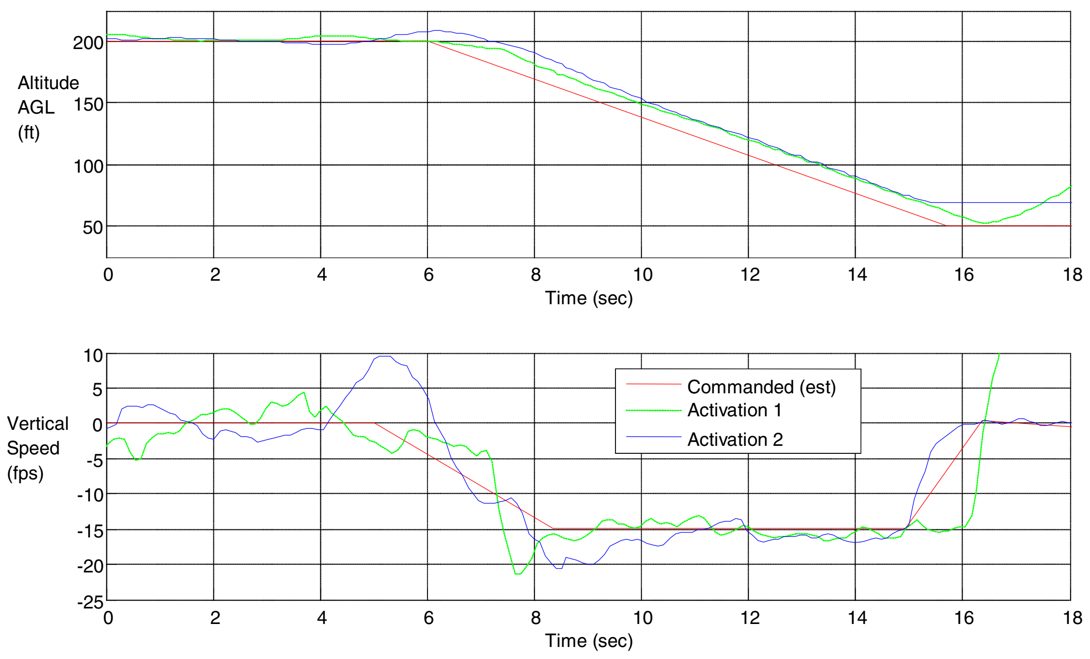


Figure 19. Descent and Landing Tracking Performance

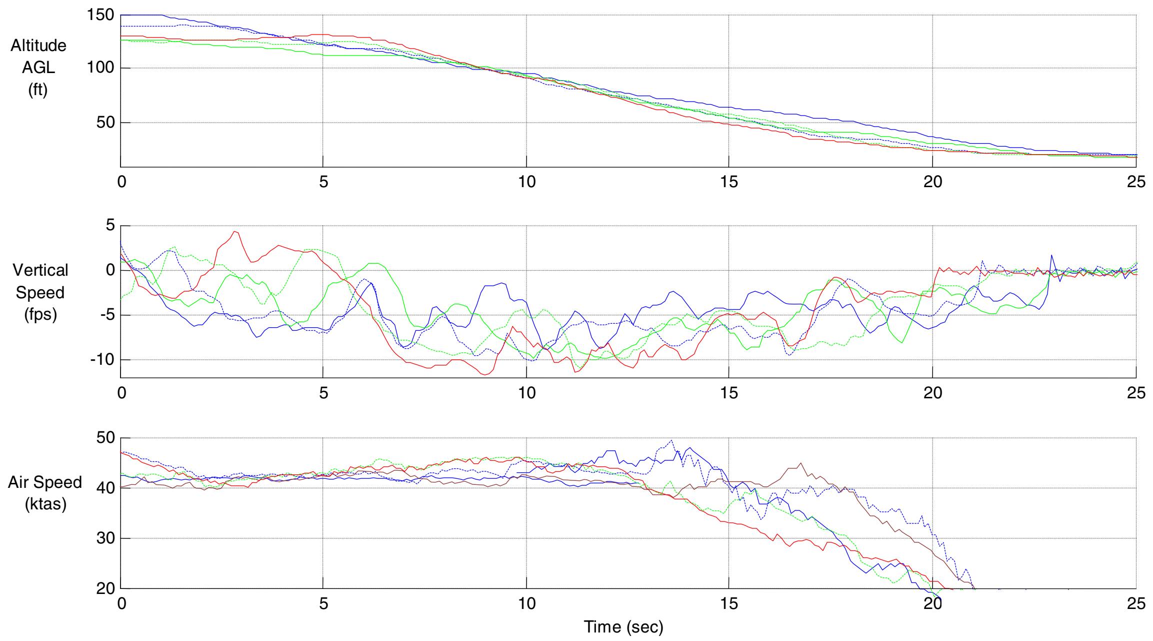


Figure 20. Flight Test Experiment Landing Profiles
(Top) Altitude profiles for various landings, as reported by K_{ekf}
(Middle) Vertical speed profiles, (Bottom) Airspeed profile.

REFERENCES

[1] C. Ippolito, K Al-Ali, "Topological Constructs for Automatic Reconfiguration of Polymorphic Control Systems", AIAA Infotech@Aerospace 2007 Conference and Exhibit, AIAA-20007-2832, May 2007

[2] S. Joo, K. Al-Ali, C. Ippolito, Y. Yeh "Towards Autonomous Fixed-Wing UAV landing: A Vision Aided INS under Sensor Reconfiguration Scenario", 17th Annual IFAC World Congress, Seoul, Korea, July 2008

[3] S. Joo, C Ippolito, K Al-Ali, and Y Yeh, "Vision Aided Inertial Navigation with Measurement Delay for Fixed-Wing Unmanned Aerial Vehicle Landing", 2008 IEEE Aerospace Conference Conference, Big Sky, Montana, 2008

[4] K. Krishnakumar, J. Kaneshige, C. Ippolito, R. Waterman, C. Pires, "A Plug and Play GNC Architecture Using FPGA Components", AIAA-2005-7120, Infotech@Aerospace, Arlington, Virginia, Sep. 26-29, 2005

[5] P Doherty, at al. "A distributed architecture for autonomous unmanned aerial vehicle experimentation", Proceedings of the 7th International Symposium on Distributed Autonomous Systems, 2004.

[6] J. Elston, B. Argrow, and E. Frew, "A Distributed Avionics Package for Small UAVs", AIAA-2005-6984, Infotech@Aerospace, Arlington, Virginia, Sep. 26-29, 2005

[7] M. Balas and S. Frost, "An Introduction to Evolving Systems of Flexible Aerospace Structures", IEEE Aerospace Conference, March 2007

[8] C. Ippolito, G. Pisanich, and K. Al-Ali, "Component-Based Plug-and-Play Methodologies for Rapid Embedded Technology Development", AIAA-2005-7122, Infotech@Aerospace, Arlington, Virginia, Sep. 26-29, 2005

[9] Ippolito, C. and Pritchett, A. "Software architecture for a Reconfigurable Flight Simulator", AIAA-2000-4501, AIAA Modeling and Simulation Technologies Conference, Denver, CO, Aug. 14-17, 2000

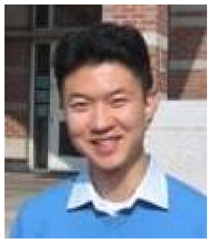
[10] Aoyama, M. "A New Age of Software Development: How Component-Based Software Engineering Changes the Way of Software Development?" In Proceedings of International Workshop on Component-Based Software Engineering, Kyoto, Japan, April 1998.

[11] Torngren, M.; DeJiu Chen; Crnkovic, I., "Component-based vs. model-based development: a comparison in the context of vehicular embedded systems.", 31st EUROMICRO Conference on Software Engineering and Advanced Applications, 30 Aug.-3 Sept. 2005 Page(s): 432 – 440, 2005.

[12] C. Ippolito, Y. Yeh, J. Kaneshige, "Neural Adaptive Flight Control Testing on an Unmanned Experimental Aerial Vehicle", AIAA-2007-2827, Infotech @ Aerospace 2007 Conference and Exhibit, Rohnert Park, California, May 7-10, 2007

BIOGRAPHY

***Corey Ippolito** is a Research Scientist and Aerospace Engineer at NASA Ames Research Center, currently Co-PI on the Polymorphic Control Systems project and runs the Exploration Aerial Vehicle lab at Ames. An MSAE from Georgia Tech in 2000, Mr. Ippolito is recipient of the NASA Award of Excellence and the NASA Team Achievement Award, with research interests that include vehicle autonomy, control reconfiguration and probabilistic methods in artificial intelligence. He is a contributing member of AIAA and IEEE, with affiliations that include the NASA Haughton-Mars Project, and the NASA Biologically-Inspired Engineering for Exploration Systems for Mars project.*



***Sungmoon Joo** is a Ph.D candidate in the Aerospace Robotics Laboratory at Stanford University. His research interests include vision aided inertial navigation, and estimation and control for differentially flat systems. Mr. Joo obtained an M.S.M.E from University of California, Berkeley in 2003. Prior to joining the ARL he worked at Korea Naval Academy as a teaching instructor in Jinhae, Korea. Mr. Joo is a member of the American Insititute of Aeronautics and Astronautics.*

***Dr. Khalid M. Al-Ali** is a Senior Fellow and Director of Research at Carnegie Mellon University's west coast campus, and founder of the Carnegie Mellon Innovations Laboratory (CMIL). He has been principal investigator, project lead, and senior scientist on projects involving advanced control systems, intelligent avionics, planetary rovers and robots, spacecraft, and autonomous exploratory vehicles for Lunar, Martian, and Antarctic missions. Dr. Al-Ali holds a Ph.D. in Mechanical and Electrical Engineering from the University of California at Berkeley. He is also President and CEO of Senseta Inc.*

***Yoo Hsiu Yeh** is a Project Engineer at Carnegie-Mellon University West Coast Campus. She graduated with a B.S.E.E. from Stanford University in 2006, and has been working with the Exploration Aerial Vehicle project at NASA Ames Research Center. She is a member of the IEEE and AIAA.*