A Dynamically Reconfiguring Avionics Architecture for UAVs

O. A. Rawashdeh[*], D. M. Feinauer[†], C.P. Harr[†], G. D. Chandler[†], D. K. Jackson[†], A.W. Groves[†], and J. E. Lumpp, Jr.[‡]

University of Kentucky, Lexington, KY 40506, USA

The goal of this research is to develop a framework for the design and implementation of reliable avionics and control architectures for UAVs. The approach is based on composing UAV control systems as software modules with explicit dependencies and possibly multiple implementations of individual modules with different resource requirements. A run-time manager analyzes the dependencies, resource requirements, and available resources to determine what tasks and which versions of individual tasks to schedule. This approach allows architectures that can adapt to both hardware and software failures by reducing the quality of the processing, the rate of processing, or by eliminating some non-critical functions instead of having to terminate the mission. This paper introduces the approach and discusses the past and future application of the techniques to UAVs developed at the University of Kentucky.

I. Introduction

THIS extended abstract presents our research in building a framework for designing avionics and control systems supporting graceful degradation. The design approach is based on a graphical software specification technique. Software *dependency graphs* (DGs) are used to specify the interaction and interdependencies among software modules. Individual software modules can be specified with alternate implementations that may require different system resources. As failures occur, a *system manager* tracks the system status and uses the dependency graphs to find a new system configuration to schedule on the available processing resources. The proposed framework also supports traditional fault-tolerance techniques such as fail-over programming, N-version redundant calculation, and voting, making it an attractive alternative for the design of a wide range of embedded control applications.

Fault tolerance is the ability of a system to continue operation despite the presence of hardware and software faults. Recognizing that no building block can be made completely error free, techniques are required to compensate or tolerate the failures. Fault tolerance is typically achieved through redundancy in hardware (and software) to enable fault detection and recovery. However, redundancy for a non-trivial system can be complex and costly in terms of size, weight, cost, and power consumption.

Graceful degradation is a promising concept to achieve higher and more cost effective fault tolerance. Graceful degradation entails a dynamic reconfiguration of the system in response to failure of hardware or software components. A fault would cause the system to lose some functionality or reduce the quality of its function as opposed to total system failure. Graceful degradation is feasible in distributed embedded systems because they typically have plentiful resources dedicated to non-critical functionality. These non mission-critical resources may be diverted to be used for critical functions when necessary.

Previous avionics systems we have developed[1] included *ad-hoc* support for failures, however, the techniques presented here allow the systems to respond to unforeseen failures and bring fault tolerance in as a primary goal early in the design process. These formal techniques are currently being applied to the development of UAVs at the University of Kentucky (UK). The BIG BLUE III Mars airplane high-altitude test platform and the UK entry in the Association for Unmanned Vehicle Systems International (AUVSI) 3rd Annual Student UAV Competition will include avionics and control systems that will support automatic reconfiguration.

[*] Research Fellow, Department of Electrical and Computer Engr., osamah@uky.edu, AIAA Student Member.
[†] Research Assistant, Department of Electrical and Computer Engr., AIAA Student Member.
[‡] Associate Professor, Department of Electrical and Computer Engr., jel@uky.edu, AIAA Member.

In Section II, an overview of the design framework is presented. Section III describes the application software specification technique using DGs. Section IV describes the runtime behavior of the designed system. Section V gives our current research status. Section VI discusses related work. Section VII is a summary of this extended abstract.

II. Architecture Overview

The hardware architecture model is shown in Figure 1. The top right block represents the system manager subsystem. The four processing element blocks (PEs) represent the processing resources of the system. Each PE can host any number and combination of input and output devices that are represented as triangles in the figure. A communication network based on the controller area network (CAN)[2] interconnects all processors and the system manager.

The system manager tracks the availability of system resources (both hardware and software) and produces new system configurations in response to faults. The system manager is composed of hardware and software that is assumed to be more reliable than the PEs. The level of trust in the system manager is justified by the fact that it is reused for many implementations of target systems and the cost of verification/validation can be amortized.

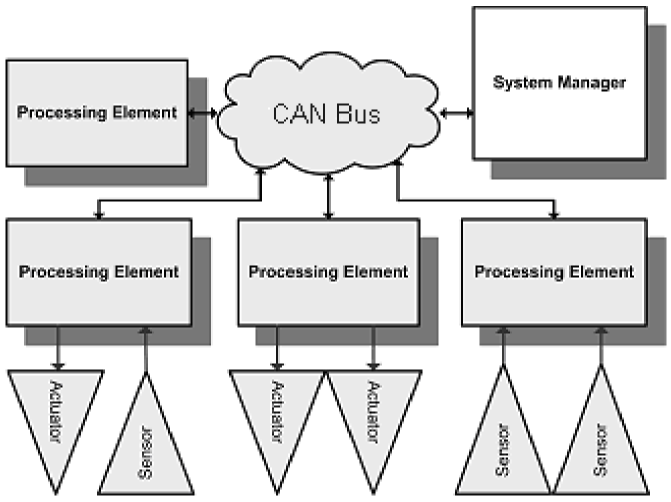

Figure 1. Hardware block diagram.

The PEs contain both a processor and a network interface. A local OS running on the processor is configured to run at least a minimal set of critical system functions and receive information from the system manager. The local OS is then responsible for executing the appropriate processes based on management messages received from the system manager. The object code of the schedulable critical software module implementations is available in the PEs local, non-volatile storage. Non-critical functions are downloaded from the system manager to PEs in the slack bandwidth of the network. A heartbeat pulse is generated by each PE and communicated to the system manager as a management message. This mechanism provides for the detection of a faulty processing element by the system manager, signaling a reconfiguration.

The I/O devices hosted by the processing elements include uni-directional input and output devices (e.g., pressure sensors and servos) and bi-directional interface devices (e.g. communication radios). Redundancy can be implemented in the form of device replicas hosted on distinct PEs or alternate I/O devices that provide similar functionality of a possibly different quality.

Communication messages transmitted on the communication bus are classified into two categories: management and application data. The system manager uses management messages to relay software scheduling information and to receive information from fault checkers as well as heartbeat messages from PEs. The data messages are used to share data and state variables among processing elements.

III. Software Model

Dependency graphs (DG) provide the basis for determining system configurations and making scheduling decisions[3]. Dependency graphs are directed graphs that specify the flow of data and the dependencies among *software modules*. A software module is a quantum of executable code that is schedulable by the OS running on a processing element. The software architecture of an application is represented as a set of dependency graphs for each system function (e.g., system output). DGs show the flow of information from input devices to output devices through a series of software modules that perform the required computations. *Data variable* nodes represent the information passed between modules. The data variable requirements of a software module are represented in the graphs by a set of interconnection symbols (comparable to logic gates). The software modules form the basic degradable units of our architecture.

Three types of symbols used to interconnect software modules with data variables shown in Figure 1. The output of the logic symbol (D) feeds either into an input of another gate or connects to a software module. The inputs to the symbols can be data variables from software modules or outputs of other symbols. The AND symbol can have one or more inputs and exactly one output. All inputs (A, B, C) to an AND are required by the object connected to its output. Secondly, the OR symbol can have one or more inputs and exactly one output. The object connected to its output does not require any of the inputs (A, B, C) to be available, but as many as possible is preferred. Finally, the XOR symbol can have two or more inputs and exactly one output. The XOR gate specifies that exactly one of the input data variables (A, B, C) is required by the object connected to its output. The highest quality (Q value) input is preferred. Hence, AND is used for required inputs, OR is used for optional inputs, and XOR is used to select one of the inputs.

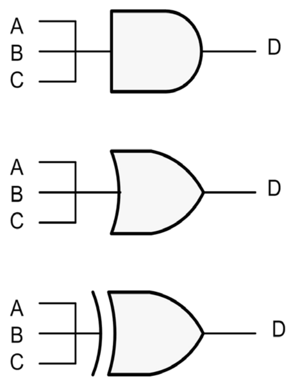

Figure 2. DG symbols.

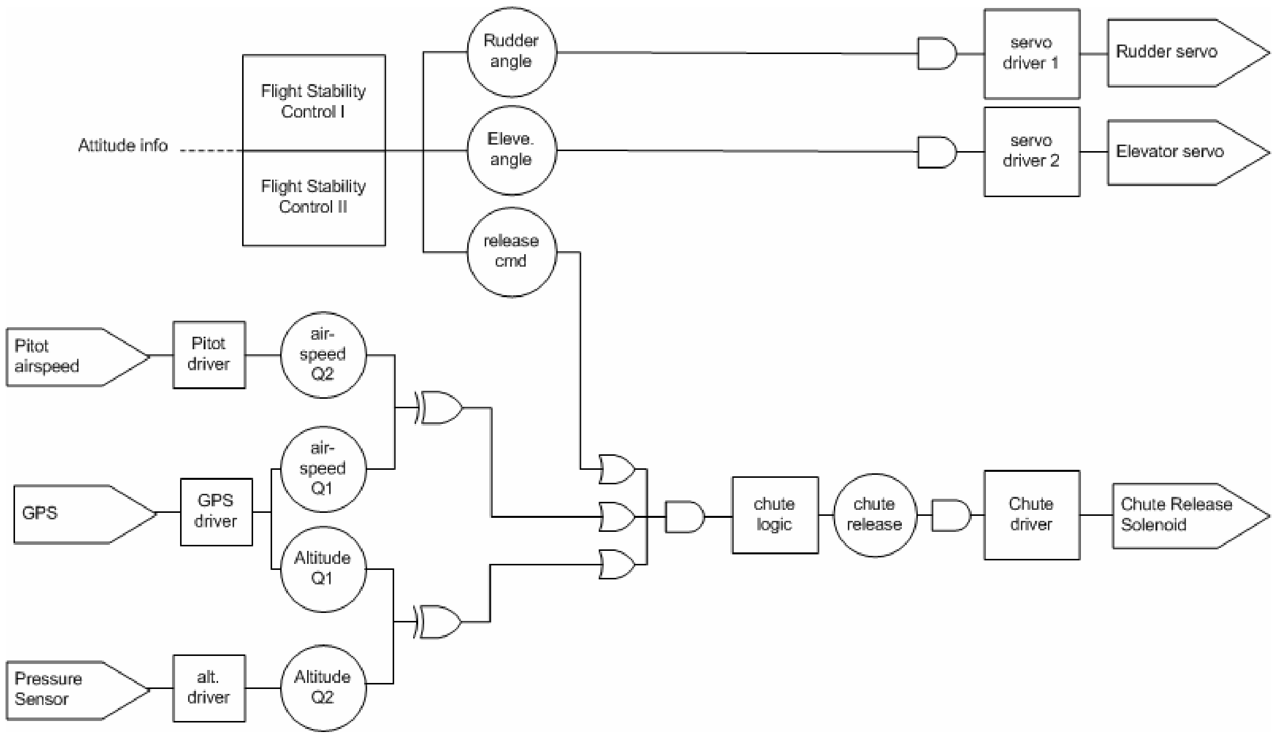

Figure 3. Partial SW dependability graph.

A partial DG is shown in Figure 3. The leaves of the graph represent I/O devices (i.e., sensors and actuators). Square nodes represent software modules and circular nodes represent data/state variables. Variable and module nodes are indexed with the number of available versions. Variables can be available in more than one quality for example. Similarly, there may be more that one version or operating mode of a software module.

IV. Runtime Operation

Fault detection is achieved by the application code and is specified through the DGs. Depending on the criticality of a software module, sensor, or actuator, a variety of fault detection techniques may be applied. For example, N-version redundancy or brute force redundancy may be implemented using a voting software module that notifies the system manager of failures. State estimation of variables may also be used to monitor data variables. Critical actuators may be monitored using sensor feedback. By monitoring the absence of heartbeat messages and monitoring the status of the software modules scheduled on a PE, the system manager can detect its failure. Furthermore, the local OS on a PE can detect violations (e.g., stack overflow, memory space violations, etc.) of software modules.

Fault handling is managed by the system manager. The fault detection mechanisms discussed above provide the manager with the condition of the hardware resources and of software modules. When a failure occurs, the manager finds a new configuration for the critical functions first. The remaining resources are then allocated to run non-critical functionality. Both *masking* and *fail-safe* fault tolerant behaviors are supported. Masking fault-tolerance entails the ability of the system to mask faults from other system components guaranteeing continued correct operation of the system. Fail-safe behavior entails placing the system in a safe mode (where all output devices are in a predefined "safe" state) if a non recoverable fault has occurred. In our framework, fault masking is achieved by system reconfiguration. Fail-stop behavior can be implemented by implanting a fail-safe version of software modules that drive output devices so the output devices are set to a "safe" state when the rest of the system fails.

The successful assignment of software modules to PEs is based on the availability of free resources on the PE (e.g., memory, processing power, and communication bandwidth) to run the module. Modules may be scheduled to PEs with slack if they do not have hard real-time deadlines. After the manager determines a new system configuration in response to failure(s), the appropriate control messages are sent to the local OS on each PE. These messages indicate which software to schedule on the PE and in which mode each software module should operate.

Figure 4. BIG BLUE II at recovery sight.

In some mission profiles, priorities of system sub-functions may change. For example, takeoff and landing phases may require more exact altitude information. To accommodate this property, the priorities of output software modules may be a function of a "mission state" variable. When the state of the mission changes, the system manager performs a reconfiguration process with the updated priorities of the system sub-functions. Therefore, the system manager can reconfigure the system due to changes in the function priorities as it does when faults are detected.

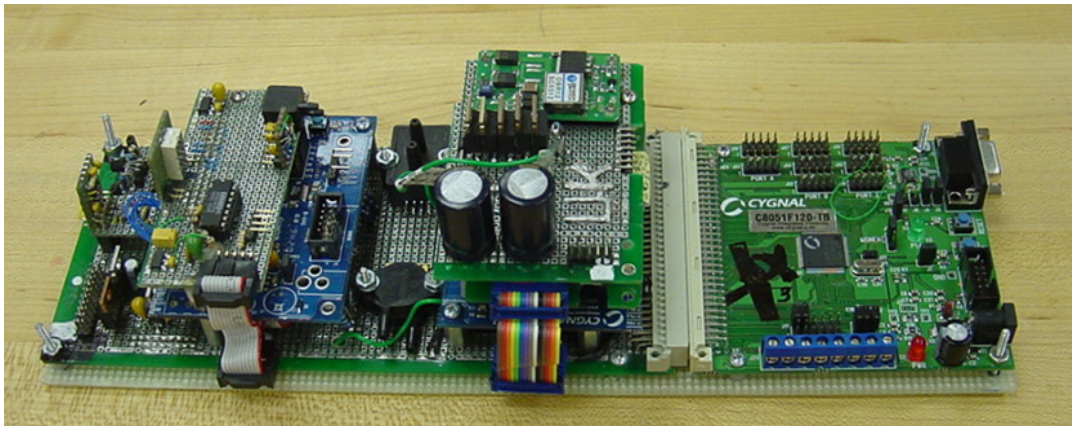

Figure 5. BIG BLUE II multi- processor avionics system.

V. Current Research Status

The methodologies described in this paper are currently being applied to ongoing autonomous UAV research projects at the University of Kentucky (e.g., Figure 4). The past two generations of BIG BLUE[4], a high-altitude inflatable wing Mars airplane test platform, have evolved to employing a multi-processor avionics system employing some traditional fault–tolerance techniques. Figure 5 shows a 3 processor avionics system for BIG BLUE II. The latest generation of BIG BLUE is currently being developed using our gracefully degrading design framework. An experiment to test a wing inflation system at over 100,000 ft. will launch in May 2005. In addition, an autonomous UAV is being developed using this framework for entry in the AUVSI 2005 Student UAV Competition. This UAV is capable of autonomous navigation and target recognition. We are also extending the architecture to integrate with a communication link to a ground station, allowing computation to transition between the two.

VI. Related Work

The software module dependency graphs used in our framework share some aspects with *success trees,* used in modeling industrial systems for fault diagnosis and risk analysis[5]. Our dependency graphs can be viewed as extended success trees. We include combinatorial gates that allow the specification of purely optional objects as well as the annotation of objects with quality values. Hence, our graphs show the flow of data instead of the binary values "operational" or "non-operational" exhibited in success trees.

Our design framework shares many similarities with the work of two other research groups applying graceful degradation to distributed embedded systems[6,7]. Our techniques focus on the problem of graceful degradation on small UAVs employing CAN bus for interconnection and we are extending the network to allow computation to migrate across wireless links to ground based computing resources.

Architectures for safety-critical reconfigurable avionics systems have been proposed[8]. These architectures use an *a priori* set of specifications (i.e., possible configurations) to simplify the assurance of critical system components. Degraded operation is facilitated on a higher level than our framework by providing complete backup sub-systems (HW and SW) with possibly different qualities.

Adaptive flight control systems are being developed that allow control systems to adapt to changes in the airframe (e.g., failure of a control surface)[9]. Our framework compliments this type of system by allowing recovery from failure of computer control components.

Standard industry practices in dealing with faults in embedded systems are incorporated in NASA's Integrated Modular Avionics Architectures Project (IMA)[10]. The aim of this project is to move away from the traditional federated design of automated flight control systems, where each function (e.g., autopilot, auto throttle, flight management) is implemented as a separate system with its own built in fault-tolerance. The IMA is a shared single processor design that is partitioned to run functions as independently as possible by promoting fault containment. The IMA uses traditional redundancy to provide tolerance in hardware faults. Also, resources such as input devices and output devices are replicated for each sub-system. The major focus of the IMA project is on partitioning a single processor such that the behavior and performance of software in one partition is unaffected by software in any other partition. Graceful degradation capability is not an explicit goal of the IMA. The only degradation it handles gracefully is that of faults in redundant hardware. In case of other faults, the focus is on fault containment and function uncoupling. However, in the implementation of the here proposed system, the problem of partitioning hardware resources to run software modules in a fault tolerant manner is very relevant.

Another NASA research project is the Mission Data System (MDS) project that was initiated in April 1998 at the Jet Propulsion Laboratory[11]. The goal of this project is to improve the software development process to achieve higher system robustness for the next generation of multi-mission autonomous deep space systems. A radical new approach is taken where actions of a system are based on reasoning instead of verdict. The reasoning is based on the effort of the system to achieve its high level mission goals in unpredictable and dynamic situations to meets its operator's intent. At the core of the MDS are state variables that describe the momentary condition of the dynamic collective system. System states include device operating modes, resource levels, device health information, temperature, pressure, etc. as well as goal-related environmental states such as motion of celestial bodies and solar flux. Each state is also accompanied by a state estimate that expresses the certainty of the state values instead of viewing them as fact. Furthermore, models are used to describe how a system's state evolves. These models explicitly express domain knowledge instead of being imbedded in program logic providing simpler verification and reusability. The MDS operates in a goal directed fashion instead of the usual linear command sequences that would require *a priori* knowledge of the states and conditions the craft will encounter. Goals are defined as state constraints with associated time intervals that express operator's intent in an unambiguous manner. So called estimators and controllers in the MDS form the goal-achievers of the system. Goal achievers elaborate to produce a goal network by dividing the goal into sub-goals that will ultimately result in actions. The goal achievers are not constrained to a certain sequence of behavior but continuously determine the best plan of action to achieve the primary goals. This built-in behavior makes fault tolerance an intrinsic feature of MDS. NASA's Mission Data System architecture is a promising approach to achieving system dependability. The limitation with the MDS approach however is that it is not trivial to decompose goals into sub-goals. Also, it is not clear how goal conflicts are resolved without causing a mission failure. The graceful degradation approach proposed here differs from MDS in that it is intended to apply the concepts on the more traditional form of a distributed embedded system with behavior-based sequential program modules.

VII. Summary

A framework for the development of dynamically reconfiguring avionics and control architectures was described. The techniques involve describing the computational tasks of the UAV as modules with explicit data flow constraints. The technique can handle both hardware and software failures making it valuable for both UAV development and in final deployment. The techniques do not require the designer to anticipate all possible failures but still allow the system to respond gracefully to their occurrence. The techniques are currently being applied to the design of avionics and control systems for two UAV projects at the University of Kentucky.

Acknowledgments

The authors gratefully acknowledge the support of the Kentucky Space Grant Consortium and Kentucky NASA EPSCoR under the direction of Drs. Richard and Karen Hackney. This research is partially funded by a NASA Workforce Development Grant and the Commonwealth of Kentucky Research Challenge Trust Fund.

References

[1]G. Chandler, D. Jackson, A. Groves, O.A. Rawashdeh, N.A. Rawashdeh, W. Smith, J. Jacob, and J.E. Lumpp, Jr., "Design of a Low-Cost Avionics, Mission Control, and Ground Communications System for a High-Altitude UAV," IEEE Aerospace Conference, to appear, Big Sky, MT, March 2005.

[2]Robert Bosch GmbH, *CAN Specification 2.0*, 1991.

[3]Osamah A. Rawashdeh and James E. Lumpp, Jr., "A Dynamic Reconfiguration System for Improving Embedded Control System Reliability," IEEE Aerospace Conference, Big Sky, to appear, MT, March 2005.

[4]A. Simpson, O.A. Rawashdeh, S. Smith, J. Jacob, W. Smith, and J.E. Lumpp, JR., "BIG BLUE: A High-Altitude UAV Demonstrator of Mars Airplane Technology," IEEE Aerospace Conference, to appear, Big Sky, MT, March 2005.

[5]Modarres, M., "Fundamental Modeling of Complex Systems using a GTST-MPLD Framwework," Proceedings of the International Workshop on Functional Modeling of Complex Technical Systems, Ispra, Italy, May 1993.

[6]Shelton, P., Koopman, P., Nace, W., "A Framework for Scalable Analysis and Design of a System-wide Graceful Degradation in Distributed Embedded Systems," Eighth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003), January 2003, Guadalajara, Mexico.

[7]M. Trapp, B. Schürmann, and T. Tetteroo, "Service-Based Development of Dynamically Reconfiguring Embedded Systems," IASTED International Conference on Software Engineering - SE 2003, Innsbruck, Austria, 2003.

[8]Strunk, Elisabeth A., John C. Knight, and M. Anthony Aiello, “Distributed Reconfigurable Avionics Architectures,” 23rd Digital Avionics Systems Conference, Salt Lake City, UT. October 2004.

[9]Christophersen, H.B., Pickell, W.J., Koller, A.A., Kannan, S.K, and Johnson, E.N., “Small Adaptive Flight Control Systems for UAVs using FPGA/DSP Technology,” Proceedings of the AIAA “Unmanned Unlimited” Technical Conference, Workshop, and Exhibit, 2004.

[10]Rushby, J., “Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurances,” NASA Contractor Report CR-1999-209347, June 1999.

[11]Rasmussen, R., “Goal-Based Fault Tolerance for Space Systems using the Mission Data System,” 2001 IEEE Aerospace Conference, March 2001, Big Sky, MT.