

DISTRIBUTED RECONFIGURABLE AVIONICS ARCHITECTURES

Elisabeth A. Strunk, John C. Knight, and M. Anthony Aiello

University of Virginia, Charlottesville, VA

Abstract

Current and upcoming avionics systems must be able to accommodate expected growing application software volume and capability. The software domain has struggled to meet increasing demands while retaining the necessary level of confidence in its appropriate operation. Meanwhile, although computing components are becoming less expensive, the fixed and operational costs of hardening them to their potential environments are not progressing with the same speed. We introduce a flexible architecture based on distribution of function and assured reconfiguration that can react to failures in both hardware and software. Reconfiguration, when its safety properties are assured, can enhance analysis capabilities for critical safety properties and reduce certification costs for much of the system. This paper outlines an architecture for assured reconfiguration, the principles of reconfiguration assurance, and the accompanying cost and safety arguments.

Introduction

The trend towards more complex avionics systems in all classes of aircraft is likely to continue for the foreseeable future, as digital electronics continue to become more powerful and less expensive. As system complexity increases, the challenge of achieving and demonstrating the necessary dependability for flight-crucial systems will become much more difficult for two major reasons. The first is the volume of software that is required and the second is the likely increase in the rate of hardware component failures [2].

Avionics systems with many millions of software source lines are being planned, and there is growing evidence that they will outstrip our already strained ability to demonstrate essential dependability properties using existing techniques. Even if there are no technical barriers to the establishment of essential properties for large systems, it is likely that vastly greater resources will have to be expended to attain these properties.

Dependability will also be difficult to achieve because system component failures will become more common as the number of components increases. Even though lower component cost means that those components can be more easily replicated, replication—and the environmental shielding that must accompany it—adds weight, leading to higher operational costs, and might be non-optimal in many circumstances.

We claim that using system-wide reconfiguration as the fundamental software system architecture is a potential solution to both of these issues. With this architecture, the system would reconfigure when system components, either hardware or software, failed, and components would be designed with this in mind. The benefit is that the majority of system components would not have to meet flight-crucial dependability requirements for all of their functionality.

Reconfiguration can address the problem of complexity by allowing a developer to target the function that must be dependable in their assurance and certification efforts. In current systems, the primary separation of safety-critical and non-safety-critical function is between applications: flight control must be more dependable than flight management, for example. But much of the additional complexity is within current applications, and it is much more difficult to clearly separate crucial and non-crucial function in the same piece of software. We are researching ways: (1) to make non-crucial software function fail-stop, so that the software either works correctly or fails in a way that does not disrupt other applications; and (2) to provide reconfiguration to alternative software specifications that has flight-crucial levels of dependability. The goal of this work is to ameliorate the dependability problems with general software complexity by providing an assured, reusable structuring mechanism for complex software reconfiguration.

Structuring the software to be reconfigurable allows dependability arguments for basic software function to be substituted for the necessary argu-

ments over the entire functional capability. Current practice, for example in flight control systems [1, 3], employs reconfiguration for exactly this purpose. However, we argue that building a system systematically to be reconfigurable in an *assured* way can provide tremendous savings when it comes to certification, and can increase our assurance that the system performs with its required dependability levels.

Reconfiguration can also help with the problems arising from increased numbers of hardware components. Much of the cost of dependable hardware comes from replication of the hardware to mask failures. We argue that infrequent failures are acceptable, as long as they do not have serious safety consequences. Designing a system to be reconfigurable enables more complex system failure responses, so that masking is relegated to being one of a set of tools that might be employed to deal with faults.

This paper summarizes our approach to overall system reconfiguration. We introduce a comprehensive system architecture designed to deal with both of the problems just discussed, and explain more clearly how it addresses them. We use a running example to illustrate our points and indicate how a reconfigurable system might be designed in practice.

The Role of Architectures

The most serious problems that arise in building large software systems are rarely in the lowest-level details of the software; rather, they are in controlling the overall structural complexity so that developers and certifiers can reason about the system. The structure that must be considered includes both application software structure, with its modules and interaction between those modules, and software distribution over available hardware together with that hardware's distribution across the aircraft. This last item is needed to prevent potential common-mode failures due to proximity. Current and upcoming avionics systems must be able to accommodate expected growing application software volume, the demands of system and application communications, and new software-intensive functional and safety capabilities. In the F-22, for example:

Mission Software (MS/W) serves as the central controller of [Integrated Avionics System] operations, interfacing to all sensors, processors, pilot controls, and displays. It manages, coordinates, and supports the overall integrated capability to search, detect, track, identify, employ weapons, and expend countermeasures against airborne or ground threats. [4]

Strong mathematical tools underlie the analyses of many aspects of aviation systems, such as structures and aerodynamics. As the artificial complexity of software reaches towards the natural complexity of physical characteristics, new software tools are required if the same level of dependability is to be met. The software domain lacks mathematical tools suitable for routine use and has struggled to meet increasing demands on its capabilities.

We believe that a combination of physical and logical architectures is a necessary starting point in developing these tools. Both are becoming more prevalent in system dependability and cost arguments. A recent shift to Integrated Modular Architectures (IMAs) attempts to counteract some of the expense of federated architectures by centralizing processing power in shielded cabinets and (theoretically) enabling standardization of Line Replaceable Units (LRUs). IMAs further complicate the software problem by requiring a detailed analysis of processor partitioning, however, and make an aircraft much more vulnerable to physical common-mode failures.

Reconfiguration can be employed in conjunction with IMAs to deal with software complexity. It becomes more powerful, however, when considering distributed hardware architectures. Distributed architectures facilitate system-level modularity and flexibility by incorporating a relatively large number of processors located close to the physical components they are controlling or monitoring [5]. They are well-suited to reconfiguration, because at their core they are naturally-partitioned, low-cost processing units that can be commanded to carry out the most appropriate function at runtime. Complete replication of function in a separate cabinet is unnecessary, and the requirements for extensive shielding and backup hardware capability can be reduced. In this paper we use as an example a system with a distributed architecture to illustrate the advantages of combining the two, although combining assured

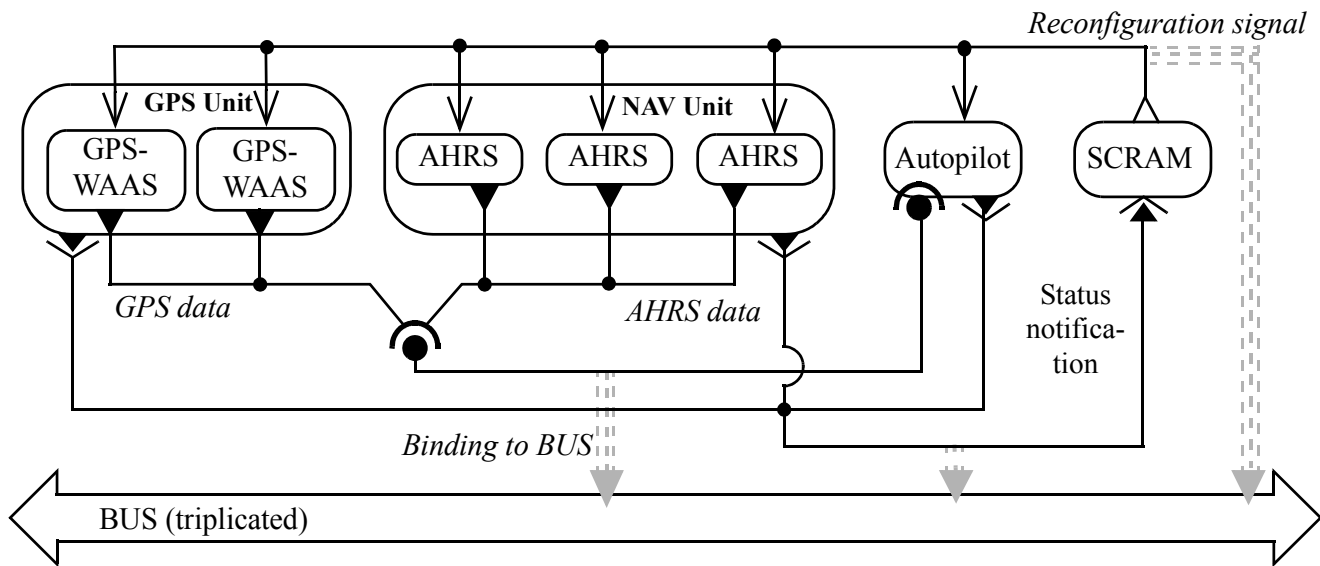


Figure 1. Example System

reconfiguration with IMAs could also be a fruitful approach to take.

Example Application

We introduce a hypothetical example avionics system to illustrate ideas as they are introduced throughout the rest of the text. The system is a small, distributed autopilot system composed of 8 components: two Global Positioning System (GPS) receivers (one primary and one backup); two Wide-Area Augmentation System (WAAS) receivers (one primary and one backup); three attitude heading reference systems (AHRs); and a 3-axis autopilot.

Each GPS receiver is paired with one of the WAAS receivers and a module that applies the differential received from the WAAS to the GPS signal. From the point of view of the system, both GPS-WAAS pairs form the GPS Unit. Each GPS-WAAS pair performs self checking; in the event that the primary GPS-WAAS pair fails, the backup GPS-WAAS pair provides the GPS Unit's output. The three AHRs function as a triple-redundant unit, the NAV Unit. The details of the voting in the TMR are hidden within the NAV Unit.

The autopilot uses the GPS Unit and the NAV Unit to control altitude and heading of the aircraft. The structure of the system is illustrated in Figure 1

using AADL, the Architecture Analysis and Design Language currently being standardized for embedded software system architectures [6].

We assume that various elements of the equipment might fail and that certain requirements exist for minimum levels of availability. If both of the GPS units fail — because of hardware or software difficulties, or because of a security attack where signals are jammed or spoofed — then the AHRs will provide enough information for the aircraft either to land at a nearby airport for maintenance or to exit the attack zone.

There is a significant difference between the standard way our example might be implemented in practice and our method for designing and implementing reconfiguration. Our method intends to give developers, certifiers, and customers confidence that the intricacies of reconfiguration requirements have been considered and met. This is done through static analysis of the applications, communications, and reconfiguration mechanism rather than through extensive mode testing on the composed product. As a corollary to this, certification arguments should become easier because it can be argued clearly that the system will function as desired even under a wide range of abnormal circumstances.

Assured Reconfiguration Architecture

We define the *system* of interest to be made up of a set of interacting *applications*. No assumptions are made about the specific functionality of these applications, although a typical set of avionics subsystems is likely. In our hypothetical example, these applications are the GPS Unit, the NAV Unit, and the Autopilot.

Each application implements a set of functional configurations. A *system* configuration is constructed from a combination of *application* configurations. The applications can communicate with one another, and a centralized, replicated controller, called the SCRAM (Subsystem Control Reconfiguration Analysis and Management) is responsible for effecting reconfiguration when necessary. It maintains state information detailing the precise configuration of each application and makes decisions about when reconfiguration is needed. The SCRAM then selects a new system configuration based on a table of values. The table tells which system configurations provide the most service to the flight crew overall under various potential environmental and failure conditions.

Application Reconfiguration

We define reconfiguration of a single application as:

The process through which [an application] halts operation under its current source specification S_i and begins operation under a different target specification S_j . [7]

In this section, we discuss a software architecture to aid application reconfiguration, and we discuss how a set of interacting applications can be reconfigured in the following section. We give a general overview of the method here; a more detailed discussion can be found elsewhere [7].

Modular Structure

In general, to reconfigure an application, it is necessary to (1) make sure the proper new function is executing, and (2) make sure that function is executing over the proper state. We assume that ensuring proper function can be subsumed by ensuring proper state. We do this by creating an extra state variable `config` representing the current configuration. A function call to `foo(x)`, then, would effec-

tively become a function call to `foo(x, config)`, where `foo(x, config)` is the proper version of `foo(x)` for configuration `config`.

It might be the case that for a particular application, separately implementing each specification to which the system can reconfigure is the most effective option. We assume, however, that it might often be more desirable to separate the application into reconfigurable modules: classes, packages, etc. (This is the more general case, since it is possible to simply specify a system with only one such module.) Each module can then be responsible for its own state, and code can be shared across implementations in many instances.

The basic structure of a single application is shown in Figure 2. The *monitoring layer* is responsible for overall supervision and control of application function. It is also the monitoring layer's job to interact with the SCRAM. Application function is accomplished by a composition of *modules*. Each module can function according to a set of potential *service levels*. The current service level to be provided by the module is specified in the *service level parameter*, an element of the module's state. An application configuration is a composition of module service levels.

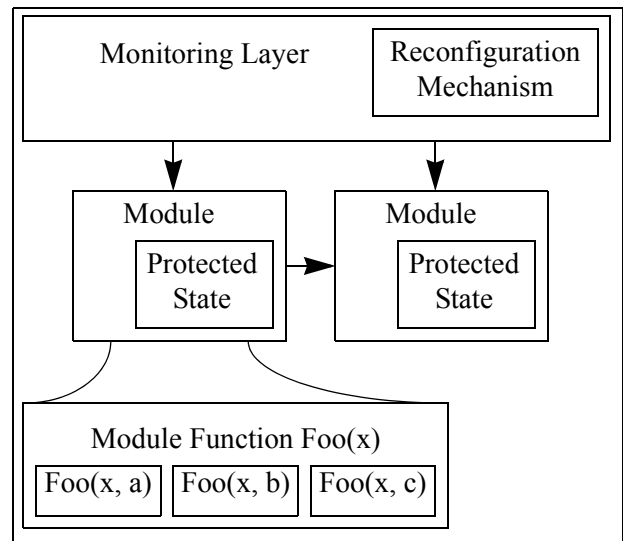


Figure 2. Single Application Structure

A unique requirement of the applications is that they are *fail-stop* [8]. This means that an application is required to operate correctly or stop and signal an error. Such behavior is much simpler to verify than

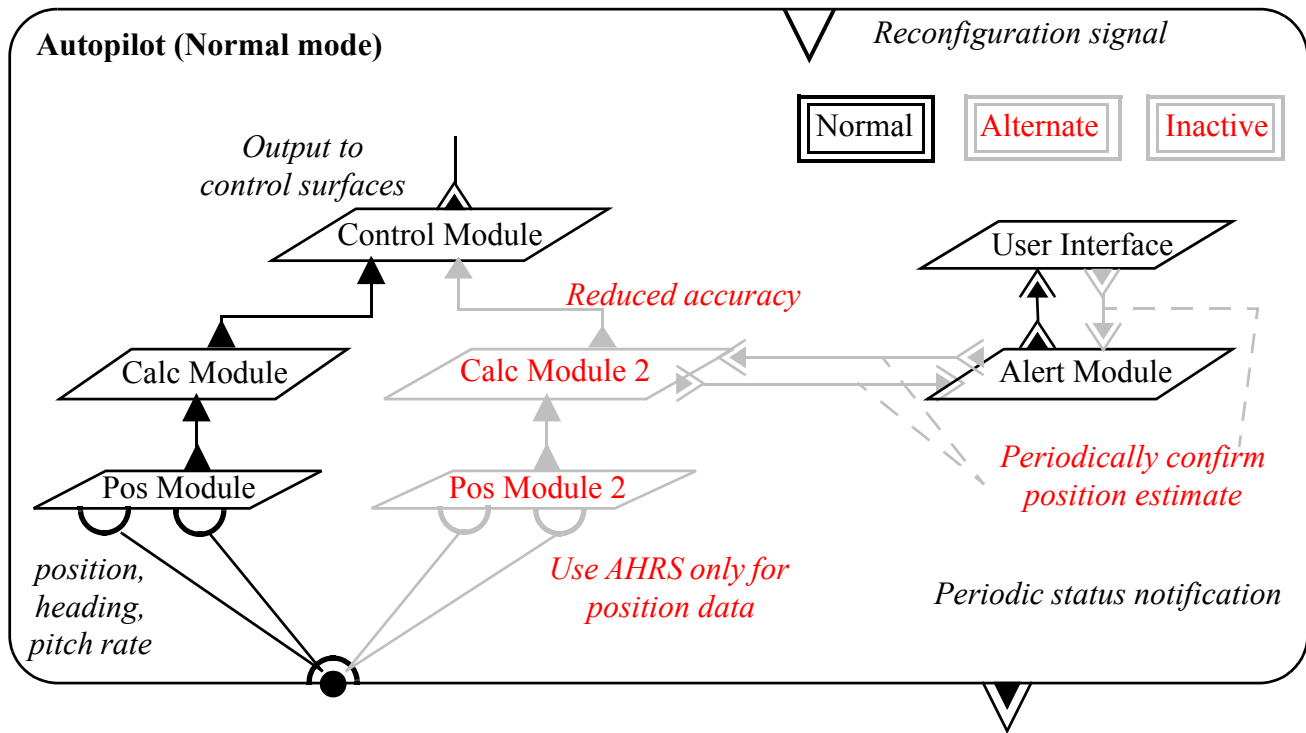


Figure 3. Autopilot Configuration View (in AADL)

complete functionality and thus presents an advantage in both development and certification. In our architecture, we push the fail-stop requirement down to the level of the module. The behavior of a module in any but the most rudimentary service levels must be fail-stop. Part of the fail-stop requirement is that module function ceases in the event of an error; another part is that the module's state may not be corrupted if an error occurs. Forcing each module to be fail-stop means that the modules' critical properties are easier to analyze, and also that the modules are simpler to compose into fail-stop applications.

If the application must reconfigure, the monitoring layer calls the *reconfiguration mechanism*, which is separate from the individual modules but can access them to call their individual reconfiguration functions. In order to effect application function, the monitoring layer would call a top-level module, whose service level parameter will have been set during initialization or reconfiguration to match the current operational specification.

In our example, the GPS Unit has three modules: one to receive and process GPS input (*GPS module*), one to receive and process WAAS input

(*WAAS module*), and one to perform differential GPS calculations (*Diff module*). (To increase reliability, the three modules are duplicated.) The NAV Unit has one triplicated module representing the standard function of the AHRS (*AHRS module*). The autopilot has four modules: one to read position information (from the GPS Unit) and rate information (from the Nav Unit) (*Pos module*), one to perform standard autopilot control calculations (*Calc module*), one to send the autopilot's outputs to the control surface actuators (*Control module*), and one to send alerts to the pilot in the event of a GPS Unit failure (*Alert module*). Most of these modules have only one service level (*Primary service level*). The *Pos module*, however, can transition to using the AHRS as its only source of position data (*AHRS Only service level*). The *Calc module*, after a fixed number of iterations, will ask the pilot to check the current position estimate (for example, through ATC communication) and correct the estimate if it is incorrect (*Reduced Accuracy service level*). The autopilot module structure is shown in Figure 3.

Reconfiguration Timing

Reconfiguration occurs through a sequence of steps:

- An error is detected.
- The application postcondition is met. In this step, all persistent application state is made consistent with what the reconfiguration mechanism must see to cause it to satisfy the precondition (starting state) of the new configuration. Each module will have a method to correct its own state, such as a rollback/rollforward mechanism or a function that resets the state to some prespecified value.
- A new configuration is selected.
- The application state is modified so that the new configuration can be put into effect. Each module will have a function to do this for its state.
- The application's transition to the new configuration is effected (in our scheme [7] this occurs automatically).
- State such as gains in a control loop is trained to be consistent with the new specification, if needed.

Our modular architecture is set up to carry out these steps in sequence. The advantage of breaking an application into reconfigurable modules is that each module can manage its own internal state, so that information hiding is employed for reconfiguration as well as for standard operation.

System Reconfiguration

The applications can communicate with one another over what is assumed to be a conventional communications mechanism such as a bus that implements an assured communications protocol. For example, a sensor consolidation application might send data to the flight control system, or multiple copies of an application running in parallel for redundancy purposes might communicate with one another to synchronize their states. The structure of the interaction and control among the applications is illustrated in Figure 1, above.

During operation, the SCRAM receives notification about any component that fails, either hardware or software. When a component fails, the SCRAM then determines what configuration the

system should be in to take best advantage of the hardware and software resources that remain. The target configuration, i.e., the one to which the system should switch, might require that applications be themselves reconfigured to provide one of their other functional specifications. This occurs in our example if the source of GPS data is lost: the autopilot then reconfigures to use the NAV Unit for both position and rate information.

The details of the forms of the various configurations, their values to the crew, and the feasible target configuration from any given state are static, i.e., they are all determined before the system is put into operation. This approach facilitates analysis of the actions that the SCRAM might take and permits stronger assurance arguments about the SCRAM itself.

As a more extensive example, suppose that in our hypothetical system, both GPS-WAAS units failed or were unable to carry out their function. The GPS Unit application would signal the SCRAM that it was no longer able to function normally. The SCRAM would then consult its table of configurations, and determine that the autopilot no longer had the necessary input precision to function normally. It would command the autopilot to transition to the Alternate specification. The autopilot then commands its Pos module to transition to AHRS Only, its Calc module to transition to Reduced Accuracy, informs the pilot of the failure, and advises the pilot to confirm periodically the position estimates by, for example, checking with ATC.

The applications' interfaces to the SCRAM ensure that the transition happens and with the appropriate timing constraints. The system mode changes are shown in Figure 4.

The SCRAM layer will have a standardized interface so that applications can be easily added to or removed from the system from a reconfiguration point of view. The standardized interface also means that the SCRAM layer can be reusable across a wide spectrum of different systems, and so the work to produce the SCRAM does not have to be repeated, and the evidence required to certify it as part of a new system will not be substantially different.

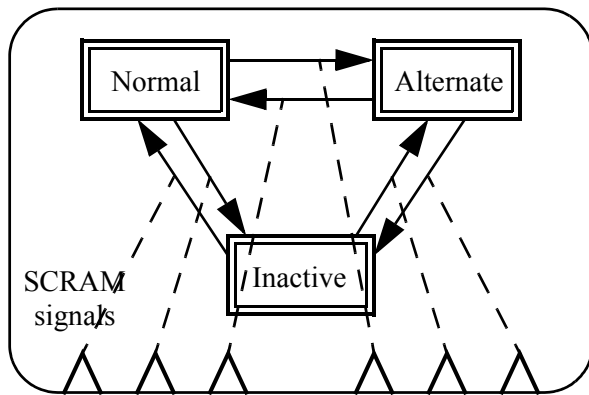


Figure 4. System Mode Transitions

Reconfiguration Specification

We require an application developer to have clearly defined the function he wants the system to execute, in the main configuration and also in any alternative configurations. Reconfiguration, then, is the act of transitioning from operating under one system specification to operating under another. We do not include provision for system learning, although within a particular specification this might be a possibility.

In order to build such a system, a developer will need the following general specification elements. For a more thorough description of the elements, see our previous explanation [9]; for a more thorough description of the timing, see our more rigorous explanation [7]. The necessary elements are:

- A: set of applications in the system
- C: set of viable system configurations
- X: set of possible transitions between configurations
- T: maximum time any member of X can take
- I: invariant on state at system level that must be either implied by individual application invariants or shown to hold during each step of reconfiguration
- E: characteristics of the system's operating environment that affect what services are most important in a particular time and place

- D: possible combinations of those characteristics that need to be considered when determining potential configurations for the system
- V: the relative value that each configuration will provide to the user under each potential combination of environmental characteristics
- P: minimum probability a specification will function, if it is called

For each application, the system survivability specification includes:

- AM: the set of modules comprising the application
- AL: for each module, the set of service levels that it provides
- AC: the set of configurations of the application, i.e., the valid combinations of module service levels
- AX: possible transitions between application configurations
- AT: worst-case transition times between application configurations
- AI: invariants over application state that must be maintained during the different transitions

A sketch of a reconfiguration specification for our example system is shown in Figure 5.

Benefits of Reconfigurable Architectures

Structuring a software system in the way we have outlined gives its designers four major benefits:

- *Limitation of critical functionality.* In order to inspect a program, or to write test cases for it, a developer must be able to understand its function. Humans are not suited to understanding the intricacies of complex designs, and so placing limits on the portion of the design that must be understood at one time is a significant path to design assurance. This is true of regulators as well as developers, and presenting a regulatory

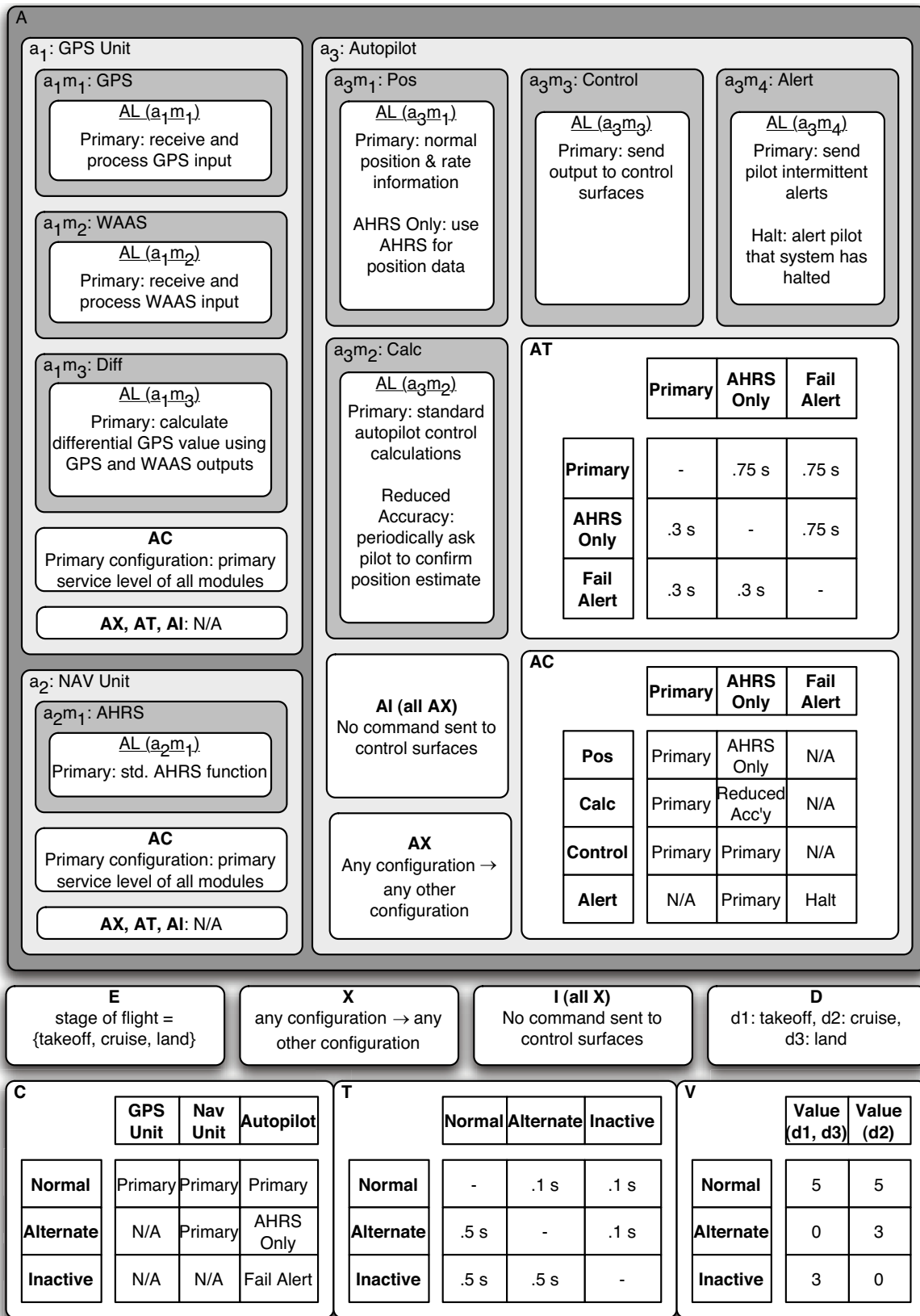


Figure 5. Overview of Survivability Specification for Autopilot System

case whose heart is controlled in size is likely to smooth the regulatory process.

- *Reduced testing costs.* In addition to controlling certification costs because the software is easier to understand, designing software to reconfigure can reduce the testing requirement on software whose function at some level is critical, but whose complete function includes conveniences the loss of which will not have catastrophic consequences. Partitioning the more complex function from the rest of the system by comprehensive checking means that checking and reconfiguration must be analyzed or tested to ensure an extremely high level of dependability, but this might not be necessary for the full function—which is the more expensive part from a certification perspective because of its size and potentially very large number of options (in the form of modes, conditions, etc.). Meeting level B requirements for an entire flight-critical application, and level A requirements for checking, reconfiguration, and very basic function, holds significant economic appeal.
- *Simplified use of COTS and legacy software.* Reconfiguration can vastly ease the use of standardized or legacy software. Porting software from one environment to another, while much cheaper than building software from scratch in many cases, can still incur major costs for recertification in the new environment. Assuring appropriate reconfiguration could allow simpler code to be thoroughly tested for the new environment, while allowing the majority of the software to be checked at runtime rather than undergoing the current necessary pre-deployment testing.
- *Support for distributed targets.* Reconfiguration is an essential requirement for systems that operate on distributed targets unless processor and communications failures can be shown to be extremely improbable. This is very difficult and expensive to do, but it is also unnecessary. By definition, distributed targets provide a computing

platform in which total loss of computing capability can be made to be extremely unlikely, even if a partial loss is not. If the software system operating on such a target is designed from the outset to be reconfigurable, it will be able to operate safely after a variety of partial platform failures.

Conclusion

The advent of yet-more-complex avionics systems and our increased dependence on them for improved safety, efficiency and functionality is inevitable. However, innovation will be severely limited if we continue to rely on centralized system architectures, even those that support IMA, and on our ability to build software that is ultra-dependable, such as is required by DO-178B for level A criticality.

In general, the solution to the problem of software complexity is not to limit the amount of function included in digital avionics. Sometimes the safety argument for refraining from including technology for which one has incomplete assurance will win out and the application will remain similar to ones built in the past. Usually, though, customer desires and other economic pressures will push for the inclusion of state-of-the-art capabilities, even if they are not accompanied by assurance arguments that are as strong as those in the state of the practice.

It is necessary, then, to develop techniques that allow desired complex systems to be built cost-effectively and with available technology, yet which permit very high levels of assurance that the system will maintain essential safety properties. In this paper, we have outlined an architectural approach in which the system goal becomes survivability rather than traditional extreme reliability. To maintain safety, the reconfiguration specification contains specifications for those functions essential to safe operation. The implementation of reconfiguration is based on the use of static analysis for the most critical functions, checking, the reconfiguration mechanism, and the SCRAM. Using static analysis in this way offers the benefits of high assurance of correct system operation, without precluding the benefits that can be gained through current state-of-the-art techniques.

Acknowledgments

It is a pleasure to thank Xiang Yin for preparation of the AADL diagrams used in this paper. This work was funded in part by NASA Langley Research Center under grants numbered NAG-1-2290 and NAG-1-02103.

References

- [1] Driscoll, K., B. Hall, H. Sivencrona, P. Zumsteg, September 2003, "Byzantine Fault Tolerance, from Theory to Reality," in Proc. International Conference on Computer Safety, Reliability, and Security, Edinburgh, UK.
- [2] Yeh, Y. C., February 1996, "Triple-Triple Redundant 777 Primary Flight Computer," in Proc. 1996 IEEE Aerospace Applications Conference, v. 1, New York, N.Y.
- [3] Sha, L., July/August 2001, "Using Simplicity to Control Complexity," *IEEE Software* 18(4):20-28.
- [4] Brower, R. W., 2001, "Lockheed F-22 Raptor," in *The Avionics Handbook*, C. Spitzer, ed., Boca Raton, FL, CRC Press, pp. 32-1 - 32-11.
- [5] Moore, J., 2001, "Advanced Distributed Architectures," in *The Avionics Handbook*, C. Spitzer, ed., Boca Raton, FL, CRC Press, pp. 33-1 - 33-13.
- [6] See www.aadl.info
- [7] Strunk, E. A., J. C. Knight, July 2004, "Assured Reconfiguration of Embedded Real-Time Software," Proc. 2004 International Conference on Dependable Systems and Networks (DSN), Florence, Italy.
- [8] Schlichting, R. D., F. B. Schneider, August 1983, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Transactions on Computing Systems* 1(3):222-238.
- [9] Knight, J. C., E. A. Strunk, 2004, "Achieving Critical System Survivability through Software Architectures," in *Architecting Dependable Systems II*, de Lemos, Gacek, and Romanovsky, eds., Springer-Verlag.