PROVABLY DEPENDABLE SOFTWARE ARCHITECTURES FOR ADAPTABLE AVIONICS

Victoria Stavridou, System Design Laboratory, SRI International, Menlo Park, CA

Abstract

Dependable architectures demonstrably possess properties such as safety, security and fault tolerance. We are interested in developing methods allowing formal demonstrations through proof that an architecture does indeed possess the desired dependability properties. We focus on architecture hierarchies as a means of enabling such demonstrations. Integrated avionics offer unprecedented generality and on-board computing utilization which are *adaptability enabling* technologies. We pose integrated avionics as a challenge problem for dependable software architectures and we propose a research agenda for solving it.

What are dependable software architectures?

Software architectures describing software products that are used to implement critical functions need to be trustworthy. *Dependability* is the property of a computing system which allows reliance to be justifiably placed on the service it delivers. The service delivered by a system is its behavior as it is perceived by its users. Dependability has qualitative and quantitative attributes and is an overall judgement about a system.

The software architecture community has made great strides towards characterizing and capturing system descriptions appropriately and towards providing linguistic support for defining families of software products, but current ADLs and their associated methodologies do not adequately address dependability. When software products are deployed in a high integrity system, their dependability profile is key to the survivability of the system. It is desirable to have developers of critical systems also benefit from software architecture technology. For example, an autopilot software producer would like to be able to derive, from a single abstract specification of a dependable software architecture, implementations for a number of aircraft variants. The derivation process would be supported by tools for applying dependability-preserving transformations that make abstract specifications more concrete. So, these implementations would be known to share dependability properties *as well as* functionality. Such dependability-preserving transformations are key not only for resource effective and timely development, but also for assurance and certification. Enriching software architecture descriptions by including dependability attributes will enable and facilitate the reuse of not only software components but certification data *as well*. In critical systems, the cost of assurance and certification is comparable with the development cost.

Dependable architectures demonstrably possess properties such as safety, security, and fault tolerance. It is our objective to produce methods allowing formal demonstrations through proof that an architecture does indeed possess the specified dependability properties. One important limitation on the utility of applying typical formal methods to reasoning about architectures is the purely informal connection between the mathematical models that are analysed and the system being modeled. If formal analysis of a model reveals the presence of a flaw, it is generally easy to determine whether that flaw is present in the implemented system. But it is highly desirable to use formal methods to establish that the software is free of certain types of flaws, such as failing to meet safety, security, or fault tolerance objectives. Security properties typically state that certain sorts of error cannot occur. For example, a security property might state that certain sorts of communication within the system — such as flow of restricted information to a component without adequate clearances — cannot occur. Tools supporting various formal methods can be used to prove that any correct implementation of the abstract mathematical model has these properties. However, there is no guarantee that the actual software correctly implements the model. The problem of gaining confidence in the correctness of the implementation is especially acute in the case of dynamic, dependable architectures, where exhaustive testing of architectural configurations is frequently prohibitively expensive, when not theoretically infeasible.

Integrated Avionics: A Challenge Problem for Dependable Software Architectures

Avionics are typically based on *federated* architectures where each function (flight management, navigation, stores management, engine management and so on) is implemented in a largely self-contained way with only narrow interfaces to other functions. This separation ensures that functions have independent failures, and the narrow interfaces limit the propagation of failures between functions. Thus, the federated architecture provides a strong natural partitioning that enforces fault containment.

However, federated architectures are profligate in their use of hardware resources since each function must provide its own fault tolerance and I/O, which results in high development, acquisition and maintenance costs. Furthermore the resulting system is inflexible in that it is hard to add new functionality and shift resources between functions making adaptivity implausible. It is also wasteful because it fails to extract maximum performance or fault tolerance from a given set of hardware resources. For example, three triple modular redundant functions use a total of nine processors but tolerate fewer total faults than a single ninefold redundant system.

Integrated avionics have been hailed as the most important innovation in avionics in 20 years [2]. They offer unprecedented generality and on-board computing resource utilization to the degree that dynamic, in-flight software reconfiguration across the architecture to optimize safety and mission characteristics becomes a distinct, if distant, vision. The most formidable obstacle to realizing this vision is the need to demonstrate that a centralized architecture failure cannot bring about simultaneous loss of functions utilizing shared resources. Fault tolerance is applied to overcome this problem, primarily through the provision of redundancy and high integrity monitoring. A more subtle issue, and one that is a significant impediment to the certification of integrated avionics, is the fact that since functions with different criticality are processed in one cabinet there is great potential for undesirable interactions between separate avionics functions. The current lack of technology allowing convincing demonstrations of safety, fault tolerance, and noninterference is delaying the deployment of integrated avionics by about 15 years in military aviation and undoubtedly many more years in the civil sector.

Integrated avionics is of considerable interest to both civil and military aviation. In the civil aviation IMA concept [9] separate avionics functions share a common hardware platform and the associated systems software (Figure 1). A variety of military integrated avionics architectures have also been proposed, the most prominent amongst these being the JIAWG Advanced Avionics Architecture [5], the Navy's Next Generation Computer Resources (NGCR) SAFENET architecture [10] and the Rapid prototyping of Application Specific Signal Processors (RASSP) portfolio of architectures [6].

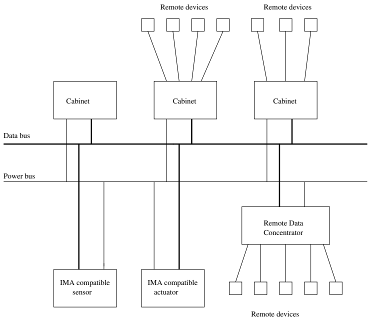

Figure 1: Integrated Modular Avionics (IMA)

What all these architectures have in common is an array of similar microprocessors, the LRMs (common line replaceable modules) housed in a cabinet. A number of these cabinets are interconnected by special buses (to ARINC 629 specification for civil aviation and to FB+, PI-bus, etc., for military aviation). The same buses connect the sensors, actuators and indicators which are treated as plug-in units. The LRMs are meant to be fully interchangeable in terms of form, fit, function and interface between different platforms, sufficiently general-purpose to be applicable across the full range of avionics functions and field-replaceable. The required architecture characteristics include interoperability, interchangeability, technology transparency, use of COTS, functional and physical integration, and an open system architecture.

The modular computer resource assets that are integrated into the platform to implement data, signal and other central processing functions required in an integrated avionics suite are called *core avionics*. Core avionics include the architecture, the backplane, the data buses, the packaging, the power supplies and the processors. The offensive, defensive and mission management functions of the avionics suite, such as sensors, communications, navigation, identification/communications security, inertial navigation are called *mission avionics*.

Getting the most out of existing defense technology with integrated avionics: An example

Maintaining weapon stores management systems on current aircraft is an ongoing challenge. Obsolete hardware and poor software engineering practices constrain the implementation of timely, cost-effective upgrades on platforms ranging from the bomber to the armed helo.

The mission need for Armed Helo evolved from the Gulf War. The Navy realized that surface forces lacked the ability to strike rapidly at the small craft/patrol boat threat in the littoral environment. Arming Naval helos with Hellfire missiles was determined as an ideal combination for combating littoral threats. The H-60 Armed Helo IPT has been working to integrate the H-60 aircraft with the Forward Looking Infrared Radar (FLIR) system and the Hellfire missile as a package that can ultimately convert the helicopter's tasks to include antisurface and antisubmarine missions. The Armed Helo development concept is to maximize the use of Non-Developmental Item (NDI) Mission Critical Computer Resource (MCCR) hardware and combinations of NDI, modified and new development MCCR software. The team is currently completing the first of three phases of the program. Phase 1 is providing the SH-60B community with eight quick-response FLIR/Hellfire Rapid Deployment Kits, which consists of the N299 quad rail Hellfire launcher, stores management unit (SMU), nose mount for FLIR, associated software, and power control. The SMU is a unique system component which serves as an interface between the aircraft and the M299 launcher. The SMU receives the aircraft data from a 1553B bus and converts it to the Mil-Std-1760 bus for the launcher. The greatest benefit of the SMU is the capability to fire other Mil-Std-1760 weapons (most forward firing weapons in the Carrier Battle Group inventory) *with a software change*. The second phase will test the integration of a similar, yet more capable system on the HH-60H. The third phase, called Core B, will test the integration of the HH-60H system on the SH-60B, ultimately providing the fleet with numerous, very capable mission kits. The Armed Helo Program has a clear commitment to computing resource reuse. Integrated avionics architectures take this vision further by enabling adaptivity on reusable computing platforms. The SMU is a complex, evolving system which is typical of the functions likely to be fielded on integrated architectures. It is a good testbed for exercising dependable software architecture theories, methods, and tools.

Enabling new defense technology with integrated avionics: An example

Uninhabited Air Vehicles (UAVs) will play an ever increasing role in future military activities. The use of reconfigurable, cooperating, multi-vehicle systems (MVSs) for diverse and flexible missions requires that individual UAVs share responsibility for safe coordination amd mission execution, including flight planning, scheduling, trajectory conflict probing and sensor data fusion. MVSs are required to support a wide variety of missions and platform configurations. Each mission will impose unique requirements on deployment of the cooperating UAVs and on their communication. For every mission, the activity required from each UAV will vary in the functions performed and the degree of human interaction required. This, in turn, requires a high degree of flexibility from the UAV avionics systems.

Currently, the ability of UAVs to jointly achieve shared mission objectives is limited by a number of factors [1]:

- The limited degree of cooperation supported by their on-board avionics.
- The reliance on the execution of a mission plan programmed into the vehicle prior to launch.
- The fairly rudimentary flight control systems allowing only a small number of maneuvers.
- The reliance on a single control mode which means they cannot adapt if that mode is no longer viable because of system malfunction or enemy action.

Highly cooperating, highly reconfigurable, adaptable UAVs require extreme flexibility in their on-board avionics. The hardware that was performing navigation in one instant may be required to perform flight control or trajectory planning in the next. This flexibility can only be achieved with reconfigurable integrated avionics. Integrated avionics systems are totally dependent on software to customize the set of standard hardware building blocks for specific applications. Thus, without integrated avionics, a whole class of software-enabled control systems, including cooperating, reconfigurable UAVs, are impossible to build.

Building and fielding integrated avionics

Building integrated avionics presents a number of challenges in terms of dynamic, real-time scheduling and resource allocation. However, the most formidable technical challenge is to demonstrate that safety has not been compromised by shifting from federated to integrated architectures. Hence, in civil aviation, where extremely stringent safety certification is required, only very conservative uses of integrated avionics are planned.

In military aviation, the ASAAC group [4] is planning the use of reconfigurable avionics while the aircraft is on the airfield. A set of pre-certified "blueprints" provides the reconfiguration executive with logical representations of the mission avionics, the overall system, and the resources available to the system in a standard format. The application blueprint describes the requirements of each mission avionics function in sufficient detail for the reconfiguration executive to allocate the necessary resources and control. The system blueprint embodies system integration decisions made at design time, including details of which mission avionics functions can use which resources. This can be thought of as the set of acceptable functional-to-physical architecture mappings. The resource blueprints describe the hardware capabilities. The standard interfaces form the basis for the mission avionics software and the operating system to be portable and independent of the underlying hardware.

The next step, which is required for cooperating, reconfigurable UAVs is full, dynamic reconfiguration in flight in response to environment or mission changes. In such unpredictable settings, unexpected scenarios, such as subsystems or whole vehicles failing, will arise. In order to continue providing acceptable service, the system must be able to detect and recover from a variety of faults.

The aim of fault tolerance is to continue providing the required service despite the presence of faults. For instance, the ASAAC fault tolerance concept is driven by the requirement to achieve full mission capability for 150 flying hours without maintenance. Even with significant improvements in component reliability, it is not considered possible to satisfy this goal, or the much more ambitious indefinitely sustained UAV flight, without the provision of some redundancy in the system. The fault tolerance concept must be capable of handling faults which occur because of battle damage, bird strike, and catastrophic failure of unrelated equipment as well as simple random component failure.

Integrated avionics support two approaches to fault tolerance, *replication* and *reconfiguration*, both of which rely on the presence of additional resources in the system over and above those required to perform the avionics functions. Replication is the traditional approach to safety-critical applications, where multiple channels are cross monitored to detect faults, and a voting system is used to identify and isolate failed components. A quadruplex flight control system is a typical example. Reconfiguration achieves fault tolerance by providing a pool of spare resources which can be used to replace failed components of the same type anywhere within the permitted reconfiguration boundary. It is anticipated that reconfiguration will be the most common method used to achieve fault tolerance for mission-critical systems. In either case, it is possible to provide graceful degradation when the level of resources available falls below that required for full functionality.

Architecture hierarchies

It is very common to describe a single architecture, or related class of architectures, at multiple levels of abstraction and from a variety of perspectives. The advantage of doing so is ob-

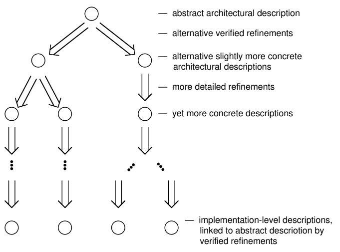

Figure 2: A correct architecture hierarchy

vious. Proving that an architectural property of interest holds at an abstract level is much easier than proving that it holds at a more concrete level. A property that is easy to prove from a data-oriented description may be difficult or impossible to prove from a function-oriented description at a similar level of abstraction.

Another motivation for having several architectural descriptions, rather than just one, is to help fill the conceptual gap between a very abstract description of the architecture and its fully concrete implementation. Often, it can be difficult to determine whether an abstract architectural description is accurately describing the implemented architecture. Unless we have confidence that the description is accurate, there is no reason to believe that properties of the description will be true of the system. More concrete architectural descriptions also provide more guidance to system implementors and maintainers.

For those reasons, we have developed an architecture description language, called SADL [8], that represents collections of architectural descriptions as *hierarchies*, with each description linked to others by interpretation mappings that have been shown to guarantee consistency of the collection, as shown in Fig. 2.

Often, it is useful to introduce a branching in the hierarchy and explore alternative implementations. Branching hierarchies can be thought of as defining a family of architectural variants. Defining variants that differ with respect to dependability properties is particularly useful. Achieving higher levels of dependability typically requires paying a price — such as increased development cost or decreased system speed — and the price that is perfectly appropriate for one application may be too high for another. Rather than making every user of an architecture pay the cost of dependability, variants that provide appropriate levels of security, fault tolerance, safety, and so on, can be developed and used where required.

An example of a family of architectures having varying dependability properties can be found in our formalization of secure variants of the X/Open Distributed Transaction Processing (DTP) standard architecture [7]. At the most abstract level of the hierarchy is a dataflow description of the X/Open standard [12] that also correctly describes the secure variants. Below this level, the hierarchy splits into secure and nonsecure branches. The nonsecure branch leads to a variety of valid implementations of the original X/Open standard. The secure branch adds certain general constraints on dataflow security, and then further branches to create variants suited to different applications. One branch describes architectures suited to applications where all components are multilevel secure, another branch describes architectures for the case where the components are single-level secure and the level of every component is the same, and yet another branch describes architectures for the case where single-level secure components vary in level.

So, current SADL technology provides a solution to the problem of relating abstract security models to concrete system implementations. A key objective in future research is to deal with other dependability properties — fault tolerance, safety, timeliness, and so on — in the same fashion. Overall system functionality and performance measures can vary radically as dynamic system architectures evolve. One sort of change is alteration or reallocation of functionality among components and connectors, with no change to connections as for example happens during a reconfiguration of an integrated avionics architecture. In this case, the structure of the architecture remains the same, while the allocation of function to structure changes. Alteration in architectural structure — adding or deleting components and connectors — is another sort of change.

Reasoning about Dependability Properties

SADL architecture description hierarchies provide a way of bridging the gap between abstract architectural models and their concrete implementations. If a hierarchy is developed using only refinement patterns that have been proven to preserve certain dependability properties, then the most concrete description in the hierarchy must have all the dependability properties of the more abstract descriptions. If it can be shown that some abstract description has a desirable dependability property, the most concrete description must also have that property. Since it is easy to verify that the most concrete description matches the actual implementation — because the most concrete description is based on the architectural constructs employed in the implementation — confidence that the property holds of the implementation is easy to obtain.

So, if an architectural description hierarchy is developed using verified refinement patterns, the proofs of the patterns guarantee that a certain class of properties is preserved. If it can be shown that one of the properties in that class holds at any level of the hierarchy, it must hold at every lower level, right down to the implementation level, as well. Thus, given a property of interest, the question naturally arises: *Does this property follow from the description at some abstract level in the hierarchy?* For some properties, like the security properties in the X/Open example mentioned above, the answer can be determined immediately by examining the abstract description. In other cases (including fault tolerance, safety and timeliness properties), complicated, error-prone reasoning may be required.

It is in these latter cases that an interface between SADL and theorem proving tools would be invaluable. An attempt to produce a formal derivation of the desired property from a description often reveals subtle errors in the argument, and in the specification itself. (Examples of incorrect informal proofs of properties of real-time system behavior revealed by the application of formal methods abound in the literature.) If the attempt is ultimately successful, confidence that the description in fact entails the property is substantially increased. For this reason, we are interfacing SADL with the PVS verification system.

Proving integrated avionics software architectures dependable

Partitioning is the process of allocating functions to computing resources. The primary partitioning requirement is that each function must not be allowed to interfere with the execution of any other function. In an integrated avionics architecture, there are three levels of partitioning: between different processors, between different partitions on the same processor, and between different processes on the same partition. High-level partitioning is very restrictive and profligate in its use of resources, but easier to achieve and demonstrate. Lower level partitioning is much more difficult to demonstrate since functions in different partitions or processes are sharing resources such as the processor, memory and communication channels.

A function could interfere with others by overruning its scheduled time, accessing the wrong memory location, or sending information to a function with which it should not communicate with. If a failure does occur within a partition, its effects must be contained; each partition should act as a fault containment region. Currently, advanced features of integrated avionics architectures cannot be fully exploited because of the onus they place on partitioning. These include *mixed criticality functions* in the same partition and the use of *reconfigurable systems*. Implementing a noncritical function in the same partition as a safety-critical one requires demonstrating that the former does not interfere with the latter. In-flight reconfiguration is currently unacceptable because of the lack of reasoning technology demonstrating the dependability of the new configuration and the switchover. This disables two significant advantages of integrated architectures — fault tolerance via reconfiguration and graceful degradation.

Achieving partitioning

High-level partitioning can be achieved by physically separating the computing modules and using independent power supplies and buses for duplicates. This provides protection against random hardware failures but not against systematic software failures. Low-level partitioning involves:

Scheduling Each cabinet contains several processors on which various functions may be executing concurrently. Each function on a processor is allocated some resources, and the operating system scheduler monitors the functions to ensure that they respect the schedule. Scheduling of functions on a single processor is synchronous; scheduling functions across more than one processor is more difficult since the processors run asynchronously and each one has its own scheduler. Infinite waiting becomes an issue.

Memory partitioning Each memory location should be written to by one function but may be read by many functions. Read access may be limited to a group of functions. Memory access policy is enforced by the memory manager.

Communication channel partitioning Each function on a processor communicates with the backplane bus via operating system communication services. These services should ensure that a function can only send messages to, and receive messages from, equipment with which it is expected to communicate. This limits the area over which a failure could cause invalid messages to be sent thus enforcing fault containment.

Integrity-level requirements All the software and the hardware that supports partitioning, including the scheduler, memory manager, and communication services of the operating system, must be developed to at least the level of integrity of the most critical function executing in the partition that they affect.

Certification authorities require evidence that partitioning failures have been analysed and that each of their effects has been allocated a severity and a frequency. These incident risks are used to determine the appropriate integrity levels to mitigate against failures. The overall requirement is that the functions implemented in a partition are adequately isolated and sufficiently independent from other functions. Current specification and verification technology does not support demonstrating the satisfaction of such dependability requirements. The innovations we propose here will facilitate such reasoning, thus paving the way to certification.

Approach to formalising partitioning

The key requirement for partitioning is demonstrating noninterference. We have formalized the noninterference concept for dependability applications in [11, 3]. We are currently performing an IMA based case study which will produce a characterization of the architectural prperties that guarantee that noninterference is maintained through the architecture hierarchy.

The next step is to formalize the partitioning. This will allow formal demonstrations of noninterference as a precursor to establishing performance, timeliness, safety, and fault tolerance properties. Two basic approaches suggest themselves. The glass-box approach models resources, applications, resource allocation, and access attempts with state machines. Then, the partitioning is specified as a property of instruction execution and operating system service invocation. In the black-box approach each LRM is modeled as a process with external port interfaces, and the LRM is mapped into an equivalent federated system. Partitioning is specified as a property that admits only behaviors that can be implemented on the federated system. The black-box approach closely matches SADL structural descriptions and STeP timed transition systems. It also allows for a more abstract, independent expression of partitioning requirements. We will pursue the black-box approach. The LRM and its functions will be mapped onto an equivalent federated system where each function occupies its own box. We will then specify the externally visible behavior of the LRM and the federated system behaviors. Thus, reasoning about the partitioning involves showing that the behavior of the LRM is also exhibited by the federated collection under the noninterference constraints.

The way forward

Our research agenda is to develop practical technology that can be used to prove the dependability properties of software architectures. We are particularly interested in timeliness, safety, fault tolerance, performance, and noninterference. We believe that the focus needs to be on reasoning about dependability-property-preserving transformations. Developing such transformations can be facilitated by focusing on a realistic, complex, ultra–high dependability application such as commercial or military avionics.

Under sponsorship from DARPA, the UK Ministry of Defence, and the UK Engineering and Physical Sciences Research Council, we have developed complementary methods and languages for reasoning about dependability properties and for specifying detailed architectural blueprints. We need to add several innovations to this background technology to facilitate definition and analysis of families of architectures with varying dependability characteristics. Specifically, we need to address these critical questions:

- *How do we add the dependability dimension to architecture definition and transformation?* ADLs need to be extended with appropriate linguistic support for expressing dependability constraints. They also need to be furnished with an appropriate semantics, to enable formal verification of architectural properties. The dependability constraints should be traceable through families of architectures and their respective hierarchies. Dependability-preserving transformations also need to be developed. Such transformations will typically involve expanding high level control and mode functions, elaborating interfaces as well as synchronization and communication protocols.

- *How do we reason formally about architectural descriptions?* We need to develop a methodology and the associated infrastructure to support intuitive and highly automated formal reasoning about architectures by augmenting ADLs with interfaces to existing formal reasoning systems.

- *How can we enable synergy between reasoning tools?* The current state of the art means that theorem provers are inadequate, individually, to reason about all aspects of complex applications. We need to find synergies between the best currently available verification systems by applying them to different facets of a real-world problem. This can leverage their respective powers, thus enabling reasoning about systems beyond the reach of current technology.

- *How do we put this technology to use so as to enable building otherwise infeasible systems?* As a community we need to find ways of getting our technology out there in the field. Toy examples simply do not cut it. We need a key example, such as integrated modular avionics, which is intractable by current technology, to exercise and refine our methods, languages and tools.

References

[1] S. Alexander, J. Koehler, J. Stolzy, and M. Andre. A Mission Management System Architecture for Cooperating Air Vehicles.

[2] R.P.G. Collinson. *Introduction to Avionics*. Chapman & Hall, 1996.

[3] B. Dutertre. A study of COTS and component integration for safety critical systems. Technical report, MOD Task Report, August 1997.

[4] R. A. Edwards. Allied standards avionics architecture council (ASAAC) phase I harmonised concept summary. Technical report, British Aerospace Defence Limited, Military Aircraft Division, 1994.

[5] *JIAWG Advanced Avionics Architecture (A^3) Standard, JIAWG Document No. J87-01*, December 1991.

[6] Lockheed Sanders, Hughes Aircraft, Motorola, and ISX Corporation. *RASSP Architecture Guide, Revision C, Document Number AVY-L-S-00081-101-C*, April 1995.

[7] M. Moriconi, X. Qian, R. A. Riemenschneider, and L. Gong. Secure software architectures. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 84–93, May 1997. Available at `http://www.csl.sri.com/sadl/sp97.ps.gz`.

[8] M. Moriconi and R. A. Riemenschneider. Introduction to SADL 1.0: A language for specifying software architecture hierarchies. Technical Report SRI-CSL-97-01, Computer Science Laboratory, SRI International, March 1997. Available at `http://www.csl.sri.com/sadl/sadl-intro.ps.gz`.

[9] A. Nadesakumar, R. M. Crowder, and C. J. Harris. Advanced system concepts for future civil aircraft — an overview of avionics architectures. *Proceedings of Institution of Mechanical Engineers*, 209:265–272, 1995.

[10] Space and Naval Warfare Command, Arlington VA. *SAFENET Tutorial, NGCR Users Conference*