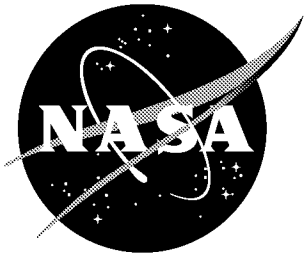
NASA/CR–2002–212130



Modular Certification

John Rushby
SRI International, Menlo Park, California

December 2002

The NASA STI Program Office ... in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

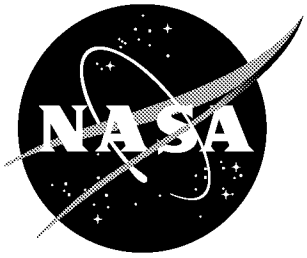
- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.
- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.
- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.
- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.
- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results ... even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***
- E-mail your question via the Internet to help@sti.nasa.gov
- Fax your question to the NASA STI Help Desk at (301) 621–0134
- Phone the NASA STI Help Desk at (301) 621–0390
- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7121 Standard Drive
  Hanover, MD 21076–1320

NASA/CR–2002–212130



Modular Certification

John Rushby
SRI International, Menlo Park, California

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681–2199

Prepared for Langley Research Center
under Cooperative Agreement NCC1-377

December 2002

Acknowledgment

For the work reported in Chapter 3, the PVS formalization of LTL was performed by Carsten Schürmann, and the PVS formalization and proof of McMillan's proof rule 3.1 was a collaborative effort with Jonathan Ford, Sam Owre, Harald Rueß, and N. Shankar.

Available from:

NASA Center for AeroSpace Information (CASI)
7121 Standard Drive
Hanover, MD 21076–1320
(301) 621–0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfi eld, VA 22161–2171
(703) 605–6000

Abstract

Airplanes are certified as a whole: there is no established basis for separately certifying some components, particularly software-intensive ones, independently of their specific application in a given airplane. The absence of separate certification inhibits the development of modular components that could be largely "precertified" and used in several different contexts within a single airplane, or across many different airplanes.

In this report, we examine the issues in modular certification of software components and propose an approach based on assume-guarantee reasoning. We extend the method from verification to certification by considering behavior in the presence of failures. This exposes the need for partitioning, and separation of assumptions and guarantees into normal and abnormal cases. We then identify three classes of property that must be verified within this framework: safe function, true guarantees, and controlled failure.

We identify a particular assume-guarantee proof rule (due to McMillan) that is appropriate to the applications considered, and formally verify its soundness in PVS.

Contents

List of Figures

Chapter 1

Introduction

Software on board commercial aircraft has traditionally been structured in *federated* architectures, meaning that each "function" (such as autopilot, flight management, yaw damping) has its own computer system (with its own internal redundancy for fault tolerance) and software, and there is relatively little interaction among the separate systems. The separate systems of the federated architecture provide natural barriers to the propagation of faults (because there is little sharing of resources), and the lack of interaction allows the certification case for each to be developed more or less independently.

However, the federated architecture is expensive (because of the duplication of resources) and limited in the functionality that it can provide (because of the lack of interaction among different functions). There is therefore a move toward integrated modular avionics (IMA) architectures in which several functions share a common (fault tolerant) computing resource, and operate in a more integrated (i.e., mutually interactive) manner. A similar transition is occurring in the lower-level "control" functions (such as engine and auxiliary power unit (APU) control, cabin pressurization), where Honeywell is developing a modular aerospace controls (MAC) architecture. IMA and MAC architectures not only allow previously separate functions to be integrated, they allow individual functions to be "deconstructed" into smaller components that can be reused across different applications and that can be developed and certified to different criticality levels.

Certification costs are a significant element in aerospace engineering, so full realization of the benefits of the IMA and MAC approach depends on modularization and reuse of certification arguments. However, there is currently no provision for separate or modular certification of components: an airplane is certified as a whole. Of course, the certification argument concerning similar components and applications is likely to proceed similarly across different aircraft, so there is informal reuse of argument and evidence, but this is not the same as a modular argument, with defined interfaces between the arguments for the components and the whole.

The certification requirements for IMA are currently under review. A Technical Standard Order (TSO) for hardware elements of IMA is due to be completed shortly [FAA01]

and committees of the US and European standards and regulatory bodies have been formed to propose guidance on certification issues for IMA. In particular, RTCA SC-200 (Requirements and Technical Concepts for Aviation Special Committee 200) and the corresponding European body (EUROCAE WG-60) have terms of reference that include "propose and document a method for transferability of certification credit between stakeholders (e.g., aircraft manufacturer, system integrator, multiple application provides, platform provider, operators, regulators)."

In this report we examine issues in constructing modular certification arguments. The examination is conducted from a computer science perspective and we hope it may prove helpful to the deliberations of the bodies mentioned above. The structure of the report is as follows. Chapter 2 provides an informal examination of the issues and proposes an approach based on *assume-guarantee* reasoning, extended from verification to certification. Chapter 3 examines the formal basis for this approach and identifies a particular assume-guarantee proof rule that is appropriate to the applications considered; the soundness of this rule is established by formal verification in PVS. Summary and conclusions are presented in Chapter 4. An appendix summarizes issues and practices in airplane certification, and outlines how these are applied to software certification.

Chapter 2

Informal Examination

The basic idea that we wish to examine is portrayed in Figure 2.1. In this diagram, X represents some function, and Y the rest of the aircraft. In the traditional method of certification, shown on the right, certification considers X and Y as an indivisible whole; in modular certification, shown on the left, the idea is to certify the whole by somehow integrating (suggested by the symbol +) properties of X considered in isolation with properties of Y considered in isolation.

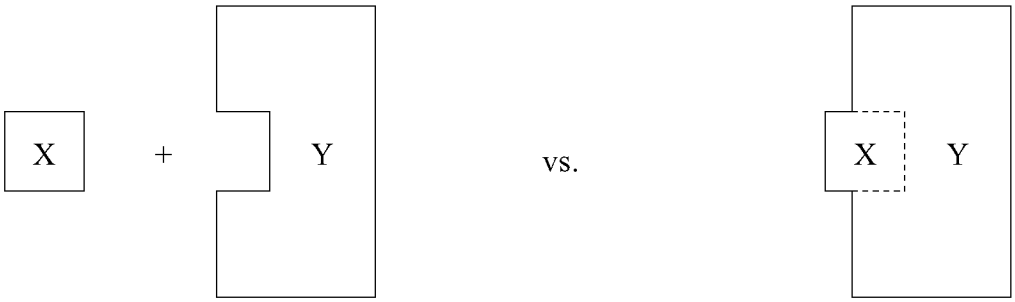


Figure 2.1: Modular vs. Traditional Certification

Many benefits would accrue if such a process were feasible, especially if the function were reused in several different aircraft, or if there were several suppliers of X-like functions for a single aircraft. In the first case, the supplier of the function could develop the certification argument for the function just once, and then contribute to the integration argument for each of its application; in the second case, the aircraft manufacturer has only to develop the integration argument for the X-like function from each different supplier. Of course, this assumes that the integration argument is less expensive, but no less safe, than the integrated argument for "X with Y" employed in the traditional method.

There is hope that modular certification should be feasible: first because there is informal reuse (i.e., modularization) of arguments concerning a function from one application to

the next, and second because it corresponds to the way systems are actually developed—as separate components with interfaces between them. Unfortunately, the hopes engendered by these observations become lessened on closer examination. It is true that systems are constructed from components and that we are used to reasoning about the properties of composite systems by considering the properties of the components and the interactions across their interfaces. The problem is that conventional design, and the notion of an interface, are concerned with normal operation, whereas much of the consideration that goes into certification concerns abnormal operation, and the malfunction of components. More particularly, it concerns the hazards that one component may pose to the larger system, and these may not respect the interfaces that define the boundaries between components in normal operation.

To give a concrete example, suppose that Y is Concorde, X is Concorde's tires. The normal interfaces between the tires and other aircraft systems are mechanical (between the wheels and the runway), and thermodynamic (the heat transfer from hot tires when the undercarriage is retracted after takeoff). The normal properties considered of the tires include their strength and durability, their ability to dispell water, and their ability to handle the weight of the airplane and the length and speed of its takeoff run and so on. These requirements of the tires flow down naturally from those of the aircraft as a whole, and they define the properties that must be considered in normal operation.

But when we consider abnormal operation, and failures, we find new ways for the tires and aircraft to interact that do not respect the normal interfaces: we now know that a disintegrating tire can penetrate the wing tanks, and that this poses a hazard to the aircraft. In a different aircraft application, this hazard might not exist, but the only way to determine whether it does or not is to examine the tires in the context of their application: in other words, to perform certification in the traditional manner suggested by the right side of Figure 2.1.

It seems that the potential hazards between an aircraft and its functions are sufficiently rich that it is not really feasible to consider them in isolation: hazards are not included in the conventional notion of interface, and we have to consider the system as a whole for certification purposes.

This is a compelling argument; it demonstrates that modular certification, construed in its most general form, is infeasible. To develop an approach that is feasible, we must focus our aims more narrowly. Now, our main concern is software, so it might be that we can develop a suitable approach by supposing that the X in Figure 2.1 is the software for some function that is part of Y (e.g., X is software that controls the thrust reversers). Unfortunately, it is easy to see that this interpretation is completely unworkable: how can we possibly certify control software separately from the function that it controls.

It seems that we need to focus our interpretation even more narrowly. The essential idea of IMA and MAC architectures, which are the motivation for this study, is that they allow software for different functions or subfunctions to interact and to share computational and communications resources: the different functions are (separately) certified with the

aircraft, what is new is that we want to conclude that they can be certified to operate *together* in an IMA or MAC environment. This suggests we reinterpret our notion of modular certification along the lines suggested in Figure 2.2.

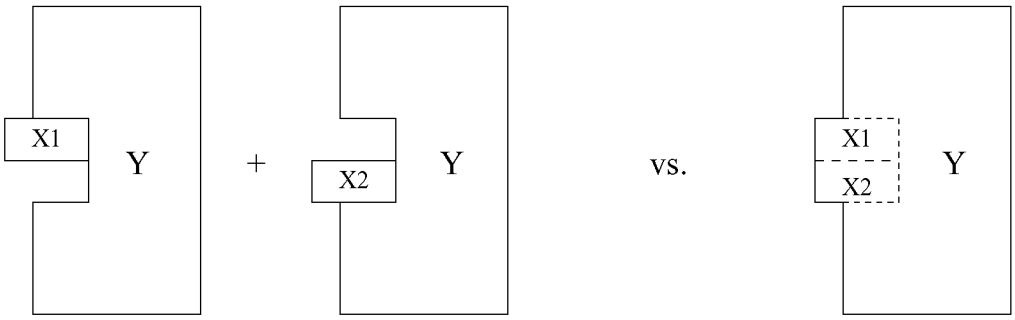


Figure 2.2: Reinterpretation of Modular Certification

The question now is: how can we certify X_1 for operation in Y without some knowledge of X_2 and *vice versa*? Suppose X_1 is the controller for the engine and X_2 is that for the thrust reverser: obviously these interact and we cannot examine all the behaviors and hazards of one without considering those of the other. But perhaps we could use *assumptions* about X_1 when examining X_2 and similarly could use assumptions about X_1 when considering X_2. Of course we would have to show that X_1 truly satisfies the assumptions used by X_2, and *vice versa*. This type of argument is used in computer science, where it is called Assume-Guarantee reasoning. Figure 2.3 portrays this approach, where $A(X_1)$ and $A(X_2)$ represent assumptions about X_1 and X_2, respectively, and the dotted lines are intended to indicate that we perform certification of X_1, for example, in the context of Y, and assumptions about X_2.

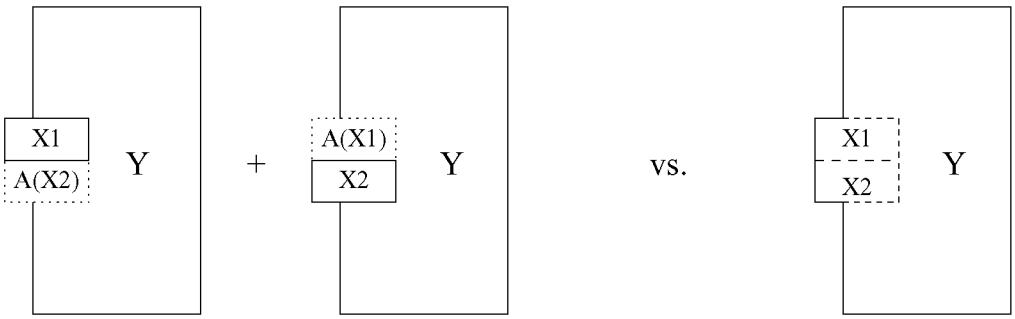


Figure 2.3: Assume-Guarantee Modular Certification

As mentioned, assume-guarantee reasoning is known and used in computer science—but it is used for *verification*, not *certification*. Verification is concerned with showing

that things work correctly, whereas certification is also concerned with showing that they cannot go badly wrong—even when other things *are* going wrong. This means that the assumptions about X_2 that are used in certifying X_1 must include assumptions about the way X_2 behaves when it has failed! This is not such an improbable approach as it may seem—in fact, it corresponds to the way avionics functions are actually designed. Avionics functions are designed to be fault tolerant and fail safe; this means, for example, that the thrust reverser may normally use sensor data supplied by the engine controller, but that it has some way of checking the integrity and recency of that data and will do something safe if that data source ceases, or becomes corrupt. In the worst case, we may be able to establish that one function behaves in a safe way in the absence of any assumptions about other functions (but it behaves in more desirable ways when some assumptions are true). There are applications and algorithms that can indeed operate under such worst-case assumptions (these are called "Byzantine fault-tolerant" algorithms). Notice that "no assumptions" does not mean "does nothing": rather, it allows any behavior at all, including that which appears actively malicious. Many avionics functions do require some minimal assumptions about other functions (for example, the thrust reverser may need to assume that the engine controller does not lose control of the engine) but we can expect that the certification of those other functions will need to ensure such minimal assumptions anyway (an engine should not go out of control, quite independently of whether the thrust reverser needs this assumption).

This analysis suggests that we can adapt assume-guarantee reasoning to the needs of certification by breaking the various assumptions and guarantees into *normal* and (possibly several) *abnormal* elements. We then establish that X_1 delivers its normal guarantee, assuming that X_2 does the same (and *vice versa*), and similarly for the various abnormal assumptions and guarantees. It will be desirable to establish that the abnormal assumptions and guarantees do not have a "domino effect": that is, if X_1 suffers a failure that causes its behavior to revert from guarantee $G(X_1)$ to $G'(X_1)$, we may expect that X_2's behavior will revert from $G(X_2)$ to $G'(X_2)$, but we do not want the lowering of X_2's guarantee to cause a further regression of X_1 from $G'(X_1)$ to $G''(X_1)$ and so on. In general, there will be more than just two components, and we will need to be sure that failures and consequent lowering of guarantees do not propagate in an uncontrolled manner. One way to achieve this is to arrange abnormal assumptions and guarantees on a series of levels, and to show that assumptions at level i are sufficient to establish the corresponding guarantees at the same level.

There is an implicit expectation here that we need to make explicit. It is the expectation that failure of X_1, say, can impact X_2 only through their respective assumptions and guarantees. Now X_1 and X_2 are software systems, so their assumptions and guarantees concern the values and relationships of various shared state variables (including those that represent real-world quantities such as time); not represented in these assumptions and guarantees are expectations that X_1 will respect interfaces even after it has failed, so X_2's private state variables will not be affected by the failure, nor will its ability to perform its computa-

tions, to access its sensors, actuators, and other private resources, and to communicate with $X_3, X_4 \ldots X_n$. These expectations are those of *partitioning*.

We have previously examined partitioning in some detail, and refer readers unfamiliar with the topic our report [Rus99]. The salient point is that architectural mechanisms external to a X_1, X_2, ... are required to enforce partitioning—for we cannot expect a failed X_1 to observe its obligations not to tamper with X_2's private variables. There might appear to be an inconsistency here: if we cannot trust a failed X_1 to observe its obligations to X_2's private variables (for example), how can we expect it to satisfy any of its abnormal guarantees? The answer is that X_1 may have several subcomponents: one failed subcomponent might (in the absence of partitioning) damage X_2, but other subcomponents will deliver suitable abnormal guarantees (for example, the software for an engine controller could fail, but a mechanical backup might then control the engine, or at least shut it down safely). In fact, partitioning is a prerequisite for this subdivision of a function into subcomponents that fail independently and therefore are able to provide fault tolerance and/or fail safety. In traditional federated systems, partitioning is ensured by physical architecture: different functions run on physically separate computer systems (e.g., autopilot and autothrottle) with little communication between them, and the subcomponents of a single function are likewise physically disjoint (e.g., the separate primary and backup of a fault-tolerant system). In an IMA or MAC system, functions and their subcomponents share many resources, so the physical partitioning of a federated architecture must be replaced by "logical" partitioning that is enforced by the IMA or MAC architecture. Constructing and enforcing this logical partitioning is the primary responsibility of the "bus" that underlies IMA and MAC architectures (e.g., SAFEbus, or TTA). Issues in the design and assurance of these safety-critical buses, and the ways in which they provide partitioning, are described in detail in another report [Rus01]. The important point is that a safety-critical bus ensures that software in a nonfaulty host computer will continue to operate correctly and will receive correct services (e.g., sensor and other data) from other nonfaulty nodes and correct services (e.g., membership and time) from the bus despite faults (software or hardware) in other nodes and hardware faults in some of the components of the bus itself. The exact types and numbers of faults that can be tolerated depend on the bus and its configuration [Rus01].

We have now identified the elements that together create the possibility of modular certification for software.

Partitioning: protects the computational and communications environment perceived by nonfaulty components: faulty components cannot affect the computations performed by nonfaulty components, nor their ability to communicate, nor the services they provide and use. The only way a faulty component can affect nonfaulty ones is by supplying faulty data, or by performing its function incorrectly. Partitioning is achieved by architectural means: in IMA and MAC architectures it is the responsibility of the underlying bus architecture, which must be certified to construct and enforce this property, subject to a specified fault hypothesis.

Assume-guarantee reasoning: allows properties of one component to be established on the basis of assumptions about the properties of others. The precise way in which this is done requires care, as the reasoning is circular and potentially unsound. This aspect is examined in a formal way in Chapter 3

Separation of properties into normal and abnormal: allows assume-guarantee reasoning to be extended from verification to certification. The abnormal cases allow us to reason about the behavior of a component when components with which it interacts fail in some way.

We say that a component is subject to an *external failure* when some component with which it interacts no longer delivers its normal guarantee; it suffers an *internal failure* when one of its own subcomponents fails. Its abnormal assumptions record the *external fault hypothesis* for a component; its *internal fault hypothesis* is a specification of the kinds, numbers, and arrival rates of possible internal failures.

Certification of an individual component must establish the following two classes of properties.

Safe function: under all combinations of faults consistent with its external and internal fault hypotheses, the component must be shown to perform its function safely (e.g., if it is an engine controller, it must control the engine safely).

True guarantees: under all combinations of faults consistent with its external and internal fault hypotheses, the component must be shown to satisfy one or more of its normal or abnormal guarantees.

Controlled failure: avoids the domino effect. Normal guarantees are at level 0, abnormal guarantees are assigned to levels greater than zero. Internal faults are also allocated to severity levels in a similar manner. We must show that if a component has internal faults at severity level i, and if every component with which it interacts delivers guarantees on level i or better (i.e., numerically lower), then the component delivers a guarantee of level i or better. Notice that the requirement for true guarantees can be subsumed within that for controlled failure.

Whereas partitioning is ensured at the architectural level (i.e., outside the software whose certification is under consideration), safe function, true guarantees, and controlled failure are properties that must be certified for the software under consideration. Controlled failure requires that a fault in one component must not lead to a worse fault in another. It is achieved by suitable redundancy (e.g., if one component fails to deliver a sensor sample, perhaps it can be synthesized from others using the methods of analytic redundancy) and self-protection (e.g., timeouts, default values and so on). Many of the design and programming techniques that assist in this endeavor are folklore (e.g., the practice of zeroing a data value after it is read from a buffer, so that the reader can tell whether it has been

refreshed the next time it goes to read it), but some are sufficiently general that they should be considered as design principles.

One important insight is that a component should not allow another to control its own progress nor, more generally, its own flow of control. Suppose that one of the guarantees by one component is quite weak: for example, "this buffer may sometimes contain recent data concerning parameter A." Another component that uses this data must be prepared to operate when recent data about A is unavailable (at least from this component in this buffer). Now, it might seem that predictability and simplicity would be enhanced if we were to ensure that the flow of data about A is reliable—perhaps using a protocol involving acknowledgments. But in fact, contrary to this intuition, such a mechanism would greatly increase the coupling between components and introduce more complicated failure propagations. For example, if X_1 supplies data to X_2, the introduction of a protocol for reliable communication could cause X_1 to block waiting for an acknowledgment from X_2 that may never come if X_2 has failed. Kopetz [Kop99] defines such interfaces that involve bidirectional flow of control as "composite" and argues convincingly that they should be eschewed in favor of "elementary" interfaces in which control flow is unidirectional. Data flow may be bidirectional, but the task of tolerating external failures is greatly simplified by the unidirectional control flow of elementary interfaces.

The need for elementary interfaces leads to unusual protocols that are largely unknown outside the avionics field. The four-slot protocol of Simpson [Sim90], for example, provides a completely nonblocking, asynchronous communication mechanism that nonetheless ensures timely transmission and mutual exclusion (i.e., no simultaneous reading and writing of the same buffer). A generalization of this protocol, called NBW (nonblocking write) is used in TTA [KR93].

There is a rich opportunity to codify and analyze the principles and requirements that underlie algorithms such as these. Codification would be undertaken in the context of the assume-guarantee approach to modular certification outlined above. That approach itself requires further elaboration in a formal, mathematical context that will enable its soundness and adequacy to be analyzed. The following chapter presents such a formal analysis for the central notion: that of assume-guarantee reasoning.

Chapter 3

Formal Examination

The previous chapter introduced assume-guarantee reasoning as the central element in an approach to modular certification. The key idea in assume-guarantee reasoning, first introduced by Chandy and Misra [MC81] and Jones [Jon83], is that we show that X_1 guarantees certain properties P_1 on the assumption that X_2 delivers certain properties P_2, and *vice versa* for X_2, and then claim that the composition of X_1 and X_2 (i.e., both running and interacting together) guarantees P_1 and P_2 unconditionally.

We can express this idea symbolically in terms of the following proof rule.

$$\frac{\begin{array}{c}\langle P_2 \rangle X_1 \langle P_1 \rangle \\ \langle P_1 \rangle X_2 \langle P_2 \rangle\end{array}}{\langle \mathit{true} \rangle \;\; X_1 || X_2 \;\; \langle P_1 \wedge P_2 \rangle}$$

Here, $X_1||X_2$ denotes the composition of X_1 and X_2 and formulas like $\langle p \rangle X \langle q \rangle$ asserts that if X is part of a system that satisfies p (i.e., p is true of all behaviors of the composite system), then the system must also satisfy q (i.e., X assumes p and guarantees q).

Rules such as this are called "compositional" because we reason about X_1 and X_2 separately (in the hypotheses above the line) and deduce properties about $X_1||X_2$ (in the conclusion below the line) without having to reason about the composed system directly. The problem with such proof rules is that they are circular (X_1 depends on X_2 and *vice versa*) and potentially unsound.

In fact, the unsoundness is more than potential, it is real: for example, let P_1 be "eventually $x = 1$," let P_2 be "eventually $y = 1$," let X_1 be "wait until $y = 1$, then set x to 1," and let X_2 be "wait until $x = 1$, then set y to 1," where both x and y are initially 0. Then the hypotheses to the rule are true, but the conclusion is not: X_1 and X_2 can forever wait for the other to make the first move.

There are several modified assume-guarantee proof rules that are sound. Different rules may be compared according to the kinds of system models and specification they support, the extent to which they lend themselves to mechanized analysis, and the extent to which they are preserved under refinement (i.e., under what circumstances can X_1 be replaced

by an implementation that may do more than X_1). Early work considered many different system models for the components—for example, (terminating) programs that communicate by shared variables, or by synchronous or asynchronous message passing—while the properties considered could be those true on termination (e.g., input/output relations), or characterizations of the conditions under which termination would be achieved. Later work considers the components as reactive systems (i.e., programs that maintain an ongoing interaction with their environment) that interact through shared variables and whose properties are formalized in terms of their behaviors (i.e., roughly speaking, the sequences of values assumed by their state variables).

One way to obtain a sound compositional rule is to break the "circular" dependency in the previous one by introducing an intermediate property I such that

$$\frac{\begin{array}{c}\langle P_1\rangle X_1\langle I\rangle \\ \langle I\rangle X_2\langle P_2\rangle\end{array}}{\langle P_1\rangle \;\; X_1||X_2 \;\; \langle P_2\rangle.}$$

The problem with this approach is that "circular" dependency is a real phenomenon and cannot simply be legislated away. In the Time Triggered Architecture (TTA), for example, clock synchronization depends on group membership and group membership depends on synchronization—so we do need a proof rule that truly accommodates this circularity. Closer examination of the circular dependency in TTA reveals that it is not circular if the temporal evolution of the system is taken into consideration: clock synchronization in round t depends on group membership in round $t-1$, which in turn depends on clock synchronization in round $t-2$ and so on.

This suggests that we could modify our previous circular rule to read as follows, where P_j^t indicates that P_j holds up to time t.

$$\frac{\begin{array}{c}\langle P_2^t\rangle X_1\langle P_1^{t+1}\rangle \\ \langle P_1^t\rangle X_2\langle P_2^{t+1}\rangle\end{array}}{\langle true\rangle \;\; X_1||X_2 \;\; \langle P_1 \wedge P_2\rangle}$$

Although this seems intuitively plausible, we really want the t and $t+1$ on the same side of each antecedent formula, so that we are able to reason from one time point to the next. A formulation that has this character has been introduced by McMillan [McM99]; here H is a "helper" property, $\Box$ is the "always" modality of Linear Temporal Logic (LTL), and $p \rhd q$ ("p constrains q") means that if p is always true up to time t, then q holds at time $t+1$ (i.e., p fails before q).

$$\frac{\begin{array}{c}\langle H\rangle X_1\langle P_2 \rhd P_1\rangle \\ \langle H\rangle X_2\langle P_1 \rhd P_2\rangle\end{array}}{\langle H\rangle \;\; X_1||X_2 \;\; \langle \Box(P_1 \wedge P_2)\rangle} \tag{3.1}$$

Notice that $p \rhd q$ can be written as the LTL formula $\neg(p \mathsf{U} \neg q)$, where U is the LTL "until" operator.[1] This means that the antecedent formulas can be established by LTL model checking if the transition relations for X_1 and X_2 are finite.

The proof rule 3.1 has the characteristics we require, but what exactly does it mean, and is it sound? These question can be resolved only by giving a semantics to the symbols and formulas used in the rule. McMillan's presentation of the rule only sketches the argument for its soundness; a more formal treatment is given by Namjoshi and Trefler [NT00], but it is not easy reading and does not convey the basic intuition.

Accordingly, we present a formalization and verification of McMillan's rule using PVS. The development is surprisingly short and simple and should be clear to anyone with knowledge of PVS.

We begin with a PVS datatype that defines the basic language of LTL (to be interpreted over a state type `state`).

```
pathformula[state : TYPE]: DATATYPE
BEGIN
  Holds(state_formula: pred[state]): Holds?
  U(arg1: pathformula, arg2: pathformula): U?
  X(arg: pathformula): X?
  ~(arg: pathformula): NOT?
  \/(arg1: pathformula, arg2: pathformula): OR?
END pathformula
```

Here, `U` and `X` represent the *until* and *next* modalities of LTL, respectively, and `~` and `\/` represent negation and disjunction, respectively. `Holds` represents application of a state (as opposed to a path) formula.

The semantics of the language defined by `pathformula` are given by the function `|=` defined in the theory `paths`. LTL formulas are interpreted over sequences of states (thus, an LTL formula specifies a set of such sequences). The definition `s |= P`[2] (`s` satisfies `P`) recursively decomposes the pathformula `P` by cases and determines whether it is satisfied by the sequence `s` of states.

[1] The subexpression $p \mathsf{U} \neg q$ holds if q eventually becomes false, and p was true at every preceding point; this is the exact opposite of what we want, hence the outer negation.

[2] PVS infix operators such as `|=` must appear in prefix form when they are defined.

```
paths[state: TYPE]: THEORY
BEGIN
  IMPORTING pathformula[state]

  s: VAR sequence[state]
  P, Q : VAR pathformula

  |=(s,P): RECURSIVE bool =
     CASES P OF
      Holds(S) : S(s(0)),
      U(Q, R): EXISTS (j:nat): (suffix(s,j) |= R) AND
                          (FORALL (i: below(j)): suffix(s,i) |= Q),
      X(Q): rest(s) |= Q,
      ~(Q) : NOT (s |= Q),
      \/(Q, R): (s |= Q) OR (s |= R)
    ENDCASES
  MEASURE P by <<
```

The various cases are straightforward. A state formula `S Holds` on a sequence `s` if it is true of the first state in the sequence (i.e., `s(0)`). `U(Q, R)` is satisfied if some suffix of `s` satisfies `R` and `Q` was satisfied at all earlier points. The functions `suffix` and `rest` (which is equivalent to `suffix(1)`) are defined in the PVS prelude. `X(Q)` is satisfied by `s` if `Q` is satisfied by the rest of `s`.

Given semantics for the basic operators of LTL, we can define the others in terms of these.

```
  CONVERSION+ K_conversion

  <>(Q) : pathformula = U(Holds(TRUE), Q) ;
  [](Q) : pathformula = ~<>~Q  ;
  &(P, Q) : pathformula = ~(~P \/ ~Q) ;
  =>(P, Q) : pathformula = ~P \/ Q ;
  <=>(P, Q) : pathformula = (P => Q) & (Q => P) ;
  |>(P, Q): pathformula = ~(U(P,~Q))

END paths
```

Here `<>` and `[]` are the *eventually* and *always* modalities, respectively. A formula `Q` is eventually satisfied by `s` if it is satisfied by some suffix of `s`. The `CONVERSION+` command is needed to turn on PVS's use of K Conversion (named after the K combinator of combinatory logic), which is needed in the application of `U` in the `<>` construction to "lift" the constant `TRUE` to a predicate on states. The *constrains* modality introduced by McMillan is specified as `|>`.

We are less interested in interpreting LTL formulas over arbitrary sequences of states than over those sequences of states that are generated by some system or program. We

specify programs as transition relations on states; a state sequence `s` is then a *path* (or *trace*) of a program (i.e., it represents a possible sequence of the states as the program executes) if each pair of adjacent states in the sequence is consistent with the transition relation. These notions are specified in the theory `assume_guarantee`, which is parameterized by a `state` type and a transition relation `N` over that type.

```
assume_guarantee[state: TYPE, N: pred[[state, state]]]: THEORY
BEGIN
  IMPORTING paths[state]

  i, j: VAR nat
  s: VAR sequence[state]

  path?(s): MACRO bool = FORALL i: N(s(i), s(i + 1))
  path: TYPE = (path?)
  p: VAR path
  JUDGEMENT suffix(p, i) HAS_TYPE path
```

A key property, expressed as a PVS judgement (i.e., a lemma that can be applied by the typechecker) is that every suffix to a path of `N` is also a path of `N`.

Next, we specify what it means for a pathformula `P` to be *valid* for `N` (this notion is not used in this development, but it is important in others).[3] We then state a useful lemma `and_lem`. It is proved by `(GRIND)`.

```
  H, P, Q: VAR pathformula

  valid(P): bool = FORALL p: p |= P

  and_lem: LEMMA (p |= (P & Q)) = ((p |= P) AND (p |= Q))
```

Next, we define the function `ag(P, Q)` that gives a precise meaning to the informal notation `<P> N <Q>` used earlier (again, the `N` is implicit as it is a theory parameter).

```
  ag(P, Q): bool = FORALL p: (p |= P) IMPLIES (p |= Q)
```

Two key lemmas are then stated and proved.

```
  agr_box_lem: LEMMA ag(H, []Q) =
    FORALL p, i: (p |= H) IMPLIES (suffix(p,i) |= Q)

  constrains_lem: LEMMA ag(H, P |> Q) =
    FORALL p, i: (p |= H)
      AND (FORALL (j: below(i)): suffix(p, j) |= P)
         IMPLIES (suffix(p, i) |= Q)
END assume_guarantee
```

[3] Note that `N` is implicit as it is a parameter to the theory; this is necessary for the `JUDGEMENT`, which would otherwise need to contain `N` as a free variable (which is not allowed in the current version of PVS).

The first lemma allows the *always* (`[]`) modality to be removed from the conclusion of an assume-guarantee assertion, while the second lemma allows elimination of the *constrains* (`|>`) modality. Both of these are proved by `(GRIND :IF-MATCH ALL)`.

Finally, we can specify composition and McMillan's rule for compositional assume-guarantee reasoning.

```
composition[state: TYPE]: THEORY
BEGIN
  N, N1, N2: VAR PRED[[state, state]]

  //(N1, N2)(s, t: state): bool =  N1(s, t) AND N2(s, t)

  IMPORTING assume-guarantee

  i, j: VAR nat
  H, P, Q: VAR pathformula[state]

  kens_thm: THEOREM
       ag[state, N1](H, P |> Q) AND ag[state, N2](H, Q |> P)
          IMPLIES
            ag[state, N1//N2](H, [](P & Q))

END composition
```

Here, `//` is an infix operator that represents composition of programs, defined as the conjunction of their transition relations. Then, `kens_thm` is a direct transliteration into PVS of the proof rule 3.1 on page 12. The PVS proof of this theorem is surprisingly short: it basically uses the lemmas to expose and index into the paths, and then performs a strong induction on that index.

```
(SKOSIMP)
(AUTO-REWRITE "and_lem[state,(N1!1 // N2!1)]")
(APPLY (REPEAT
   (THEN (REWRITE "agr_box_lem") (REWRITE "constrains_lem"))))
(INDUCT "i" :NAME "NAT_induction")
(SKOSIMP* :PREDS? T)
(GROUND)
(("1" (APPLY (THEN (INST -6 "p!1" "j!1") (LAZY-GRIND))))
 ("2" (APPLY (THEN (INST -5 "p!1" "j!1") (LAZY-GRIND)))))
```

Our first attempt to formalize this approach to assume-guarantee reasoning was long, and the proofs were also long—and difficult. Other groups have apparently invested months of work in a similar endeavor without success. That the final treatment in PVS is so straightforward is testament to the expressiveness of the PVS language (e.g., its ability to define LTL in a few dozen lines) and the power and integration of its prover (e.g., the predicate

subtype `path` and its associated `JUDGEMENT`, which automatically discharges numerous side conditions during the proof).

Chapter 4

Conclusions

We have examined several ways in which the notion of modular certification could be interpreted and have identified one that is applicable to software components in IMA and MAC architectures. This interpretation is based on the idea that these components can be certified to perform their function in the given aircraft context using only *assumptions* about the behavior of other software components.

We identified three key elements in this approach.

- Partitioning
- Assume-guarantee reasoning
- Separation of properties into normal and abnormal

Partitioning creates an environment that enforces the interfaces between components; thus, the only failure modes that need be considered are those in which software components perform their function incorrectly, or deliver incorrect behavior at their interfaces. Partitioning is the responsibility of the safety-critical buses such as SAFEbus and TTA that underlie IMA and MAC architectures.

Assume-guarantee reasoning is the technique that allows one component to be verified in the presence of assumptions about another, and *vice versa*. This approach employs a kind of circular reasoning and can be unsound. In Chapter 3, we examined formal interpretations of this technique and identified one, due to McMillan, that seems suitable. We then formalized this approach in PVS and verified its soundness.

To extend assume-guarantee reasoning from verification to certification, we showed that it is necessary to consider abnormal as well as normal assumptions and guarantees. The abnormal properties capture behavior in the presence of failures. To ensure that the assumptions are closed, and the system is safe, we identified three classes of property that must be established using assume-guarantee reasoning.

- Safe function

- True guarantees
- Controlled failure

The first of these ensures that each component performs its function safely under all conditions consistent with its fault hypothesis, while the second ensures that it delivers its appropriate guarantees. Controlled failure is used to prevent a "domino effect" where failure of one component causes others to fail also.

For this approach to be practical, components cannot have strong or complex mutual interdependencies. We related this issue to Kopetz's notion of "composite" and "elementary" interfaces. In his classic book, Perrow [Per84] identified two properties that produce systems that are susceptible to catastrophic failures: *strong coupling* and *interactive complexity*. It may be feasible to give a precise characterization of these notions using the approach introduced here (one might correspond to difficulty in establishing the property of controlled failure, and the other to excessively numerous and complex assumptions).

Interesting future extensions to this work would be to expand the formal treatment from the single assumption-guarantee for each component that is adequate for verification to the multiple (normal/abnormal) assumptions required for certification. This could allow formalization and analysis of the adequacy of the properties safe function, true guarantees, and controlled failure.

Although we have proved our assume-guarantee method to be sound, it is known to be incomplete (i.e., there are correct systems that cannot be verified using the rule 3.1). Namjoshi and Trefler [NT00] present an extended rule that is both sound and complete, and it would be interesting to extend our PVS verification to this rule.

Another extension would expand the formal treatment from the two-process to the n-process case (this is a technical challenge in formal verification, rather than an activity that would yield additional insight).

It will also be useful to investigate practical application of the approach presented here. One possible application is to the mutual interdependence of membership and synchronization in TTA: each of these is verified on the basis of assumptions about the other. Other potential applications may be found among those intended for the Honeywell MAC architecture.

Finally, we hope that some of this material may prove useful to the deliberations of RTCA SC-200/EUROCAE WG-60, which is considering certification issues for IMA.

Appendix A

Airplane Certification

Requirements and considerations for certification of software in airborne systems are described in FAA Advisory Circular 25.1309-1A [FAA88] and in DO-178B [RTC92], which is incorporated by reference into the former document (see [FAA93]). In Europe, certification is performed by the Joint Airworthiness Authority (JAA) that was set up in 1988 by the individual authorities of France, Germany, The Netherlands, and the United Kingdom. The requirements and documents of the JAA parallel those of the FAA. In particular, JAA "Advisory Material–Joint" document AMJ 25.1309 is equivalent to the FAA Advisory Circular 25.1309A cited above, and European document EUROCAE ED-12B is the same as DO-178B.

The general approach to development and certification of safety-critical systems is grounded in hazard analysis; a *hazard* is a condition that can lead to an accident. *Damage* is a measure of the loss in an accident. The *severity* of a hazard is an assessment of the worst possible damage that could result, while the *danger* is the probability of the hazard leading to an accident. *Risk* is the combination of hazard severity and danger. The goal in safety engineering is to control hazards. During requirements and design reviews, potential hazards are identified and analyzed for risk. Unacceptable risks are eliminated or reduced by respecification of requirements, redesign, incorporation of safety features, or incorporation of warning devices.

For example, if the concern is destruction by fire, the primary hazards are availability of combustible material, an ignition source, and a supply of oxygen. If at all possible, the preferred treatments are to eliminate or reduce these hazards by, for example, substitution of nonflammable materials, elimination of spark-generating electrical machinery, and reduction in oxygen content (cf. substitution of air for pure oxygen during ground operations for Project Apollo after the Apollo 1 fire). If hazard elimination is impossible or judged only partially effective, then addition of a fire suppression system and of warning devices may be considered. The effectiveness and reliability of these systems then becomes a safety issue, and new hazards may need to be considered (e.g., inadvertent activation of the fire suppression system).

The criticality of a particular component or system is a measure of the severity of the possible effects that could follow from failure of that component or system. Failure includes the possibility of performing functions incorrectly, or performing unintended functions, as well as the loss of intended functions. Design alternatives are explored in order to reduce the number of critical components and systems, and their degree of criticality. The degree of criticality associated with a particular component or system determines the degree of assurance that should be provided for it: a system whose failure could have grave consequences will be considered highly critical and will require very strong assurances that its failure will be extremely improbable.

For airplane certification, failures are categorized on a five-point scale from "catastrophic" through "hazardous/severe-major," "major," and "minor" to "no effect" [FAA88]. Catastrophic failure conditions are "those which would prevent continued safe flight and landing" [FAA88, paragraph 6.h(3)] and include loss of function, malfunction, and unintended function. Failure condition severities and probabilities must have an inverse relationship; in particular, catastrophic failure conditions must be "extremely improbable" [FAA88, paragraph 7.d]. That is, they must be "so unlikely that they are not anticipated to occur during the entire operational life of all airplanes of one type" [FAA88, paragraph 9.e(3)]. "When using quantitative analyses... numerical probabilities... on the order of 10^{-9} per flight-hour[1] may be used... as aids to engineering judgment... to... help determine compliance" with the requirement for extremely improbable failure conditions [FAA88, paragraph 10.b]. An explanation for this figure can be derived [LT82, page 37] by considering a fleet of 100 aircraft, each flying 3,000 hours per year over a lifetime of 33 years (thereby accumulating about 10^7 flight-hours). If hazard analysis reveals ten potentially catastrophic failure conditions in each of ten systems, then the "budget" for each is about 10^{-9} if such a condition is not expected to occur in the lifetime of the fleet. An alternative justification is obtained by projecting the historical trend of reliability achieved in modern jets. From 1960 to 1980, the fatal accident rate for large jets improved from 2 to 0.5 per 10^6 hours, and was projected to be below 0.3 per 10^6 hours by 1990 [LT82, page 28]. This suggests that less than one fatal accident per 10^7 hours is a feasible goal, and the same calculation as above then leads to 10^{-9} as the requirement for individual catastrophic failure conditions.

Note that the probability 10^{-9} is applied to (sub)system failure, not to any software the system may contain. Numerical estimates of reliability are not assigned to software in safety-critical systems [RTC92, Subsection 2.2.3], primarily because software failure is not random but systematic (i.e., due to faults of specification, design, or construction), and because the rates required are too small to be measured. These points are elaborated in the following paragraphs.

Failures can be *random* or *systematic*; the former are due to latent manufacturing defects, wear-out and other effects of aging, environmental stress (e.g., single-event upsets

[1] "Based on a flight of mean duration for the airplane type. However, for a function which is used only during a specific flight operation; e.g., takeoff, landing etc., the acceptable probability should be based on, and expressed in terms of, the flight operation's actual duration" [FAA88, paragraph 10.b].

caused by cosmic rays), and other degradation mechanisms that afflict hardware components, while the latter (which are sometimes called *generic* faults) are due to faults in the specification, design, or construction of the system. The probability of random failure in a system can be measured by sufficiently extensive and realistic testing, or (for suitably simple systems) it can be calculated from historical reliability data for its component devices and other known factors, such as environmental conditions. Classical fault-tolerance mechanisms (e.g., n-modular redundancy, standby spares, backup systems) can be used to reduce the probability of system failure due to random component failures to acceptable levels, though at some cost in system complexity—which is itself a potential source of (systematic) design faults.

Systematic failures are not random: faults in specification, design, or construction will cause the system to fail under specific combinations of system state and input values, and the failure is *certain* whenever those combinations arise. But although systematic failures occur in specific circumstances, occurrences of those circumstances are associated with a random process, namely, the sequence over time of inputs to the system.[2] Thus, the manifestations of systematic failures behave as stochastic processes and can be treated probabilistically: to talk about a piece of software having a failure rate of less than, say, 10^{-9} per hour, is to say that the probability of encountering a sequence of inputs that will cause it to exhibit a systematic failure is less than 10^{-9} per hour. Note that this probabilistic measure applies whether we are talking about system reliability or system safety; what changes is the definition of failure. For reliability, a failure is a departure from required or expected behavior, whereas for safety, failure is any behavior that constitutes a hazard to the continued safe operation of the airplane. This, apparently small, difference between the notions of reliability and safety nonetheless has a profound impact on techniques for achieving and assuring those properties.

First, although there may be many behaviors that constitute failure from the reliability point of view, there may be relatively few that constitute safety failures—especially of the higher severity classes. Thus, whereas reliability engineering seeks to improve the quality of the system in general, safety engineering may prefer to concentrate on the few specific failures that constitute major hazards; at the risk of reducing these complex issues almost to caricature, we could say that reliability tries to maximize the extent to which the system works well, while safety engineering tries to minimize the extent to which it can fail badly.

Second, techniques for improving reliability naturally deal first with the major sources of unreliability: that is, the most frequently encountered bugs get fixed first. There is a huge variation in the rate at which different faults lead to failure, and also in the severity of their consequences. Currit, Dyer, and Mills [CDM86] report data from major IBM systems showing that one third of the faults identified had a mean time to failure (MTTF) of over 5,000 years (and thus have an insignificant effect on overall MTTF), and a mere 2% of the faults accounted for 1,000 times more failures than the 60% of faults encountered least of-

[2] Its *environment*—the states of the other systems with which it interacts—is considered among the inputs to a system.

ten. Reliability-based approaches would concentrate on detecting and removing the faults that contribute most to unreliability (indeed, the cited data are used by Currit, Dyer, and Mills to demonstrate that random testing would be 30 times more effective than structural testing in improving the reliability of the systems concerned). The most rarely encountered faults can therefore hide for a long while under a testing and repair regime aimed at improving reliability—but if just one or two rare faults could lead to catastrophic failure conditions, we could have a reliable but unsafe system.[3] Data cited by Hecht [Hec93] indicate that such rare faults may be the dominant cause of safety- and mission-critical failures.

Third, the reliability-engineering approach can lead to concentration on the reliability of individual components and functions, whereas some of the most serious safety failures have been traced to poorly understood top-level requirements and to unanticipated subsystem interactions, often in the presence of multiple failures (Leveson [Lev86] quotes some examples and, although it does not concern software, Perrow's classic study [Per84] is still worth examination).

Elements of both the reliability and safety engineering approaches are likely to be needed in most airborne systems: although a reliable system can be unsafe, an unreliable system is unlikely to be safe in these applications. (This is true primarily because there are few safe failure modes in flight- or engine-control applications. This can be contrasted with nuclear power generation, where a protection system that shuts the reactor down unnecessarily may be unreliable, but perfectly safe.)

Just as the subtly different characteristics of reliability and safety lead to differences in methods used to achieve those properties, so they also lead to differences in their methods of assurance. Both reliability and safety are measured in probabilistic terms and can, in principle, be assessed by similar means. However, the numerical requirements for safety in airborne systems are orders of magnitude more stringent than those normally encountered for reliability. Systems designated "highly reliable" may be required to achieve failure rates in the range 10^{-3} to 10^{-6} per hour, whereas requirements for safety often stipulate failure rates in the range 10^{-7} to 10^{-12} per hour.[4] Failure rates of 10^{-7} to 10^{-12} per hour are generally considered to define the requirements for "ultra-dependable" systems. Bear in mind that these probabilities generally refer only to the incidence of safety-critical failures, and not to the general reliability of the systems concerned, and are assessed on complete systems—not just the software they contain.

The change in acceptable failure rates between highly reliable and ultra-dependable systems has such a profound impact that it goes beyond a difference of degree and becomes a

[3] With 500 deployed systems, a single serious fault with an MTTF of 5,000 years could provoke several catastrophic failure conditions over the lifetime of the fleet.

[4] Nuclear protection systems require a probability of failure on demand of less than 10^{-4} [LS93]; failures that could contribute to a major failure condition in an aircraft require a failure rate less than 10^{-5} per hour [FAA88, paragraph 10.b(2)]; the (now abandoned) Advanced Automation System for Air Traffic Control had a requirement for less than 3 seconds unavailability per year (about 10^{-7}) [CDD90]; failures that could contribute to a catastrophic failure condition in an aircraft require a failure rate less than 10^{-9} per hour [FAA88, paragraph 10.b(3)]; controllers for urban trains must have failure rates lower than 10^{-12} [LS93].

difference in kind, the reason being that it is generally impossible to experimentally validate failure rates as low as those stipulated for ultra-dependability.

There are two ways to estimate the failure rate of a system: the experimental approach seeks to measure it directly in a test environment; the other approach tries to calculate it from the known or measured failure rates of its components, plus knowledge of its design or structure (Markov models are often used for this purpose).

The experimental approach faces two difficulties: first is the question of how accurately the test environment reproduces the circumstances that will be encountered in operation; second is the large number of tests required. If we are looking for very rare failures, it will be necessary to subject the system to "all up" tests in a highly realistic test environment—installed in the real airplane, or very close facsimile (e.g., an "iron bird"), with the same sensors and actuators as will be used in flight. Furthermore, it will clearly be necessary to subject the system to very large numbers of tests (just how large a number will be explored shortly)—and if we are dealing with a control system, then a test input is not a single event, but a whole trajectory of inputs that drives the system through many states.[5] And since we are dealing with a component of a larger system, it will be necessary to conduct tests under conditions of single and multiple failures of components that interact with the system under test. Obviously, it will be very expensive to set up and run such a test environment, and very time-consuming to generate the large and complex sets of test inputs and fault injections required.

So how many tests will be required? Using both classical and Bayesian probabilistic approaches, it can be shown that if we want a median time to failure of n hours, then we need to see approximately n hours of failure-free operation under test [LS93].[6] So if we are concerned with catastrophic failure conditions, we will need to see 10^9 failure-free hours of operation under test. And 10^9 hours is a little over 114,000 years! [7]

To reduce the time under test, we could run several systems in parallel, and we could try "accelerated testing," in which the inputs to the system are fed in faster than real time and, if necessary, the system is run on faster hardware than that which will be used in actual operation. (This naturally raises questions on the realism of the test environment—particularly when one considers the delicacy of timing issues in control systems.[8]) But at

[5]The key issue here is the extent to which the system accumulates state; systems that reinitialize themselves periodically can be tested using shorter trajectories than those that must run for long periods. For example, the clock-drift error that led to failure of Patriot missiles [GAO92] required many hours of continuous operation to manifest itself in a way that was externally detectable.

[6]The Bayesian analysis shows that if we bring no prior belief to the problem, then following n hours of failure-free operation, there is a 50:50 chance that a further n hours will elapse before the first failure.

[7]Butler and Finelli [BF93] present a similar analysis and conclusion (see also Hamlet [Ham92]). Parnas, van Schouwen, and Kwan [PvSK90] use a slightly different model. They are concerned with estimating *trustworthiness*—the probability that software contains no potentially catastrophic flaws—but again the broad conclusion is the same.

[8]Its feasibility is also questionable given the fault injections that are needed to test the fault-tolerance mechanisms of safety-critical systems: experiments must allow a reasonable time to elapse after each injected fault to see if it leads to failure, and this limits the amount of speedup that is possible.

best these will reduce the time required by only one or two orders of magnitude, and even the most wildly optimistic assumptions cannot bring the time needed on test within the realm of feasibility. Similarly, quibbles concerning the probability models used to derive the numbers cannot eliminate the gap of several orders of magnitude between the amount of testing required to determine ultra-dependable failure rates experimentally, and that which is feasible.

The analyses considered so far assume that no failures are encountered during validation tests; any failures will set back the validation process and lower our estimate of the failure rate achieved. "Reliability growth models" are statistical models that avoid the need to restart the reliability estimation process each time an error is detected and repaired; they allow operational reliability to be predicted from observations during system test, as bugs are being detected and repaired [MIO87]. But although they are effective in commercial software development, where only modest levels of reliability are required, reliability growth models are quite impractical for requirements in the ultra-dependable region. Apart from concerns about the accuracy of the model employed,[9] a law of diminishing returns greatly lessens the benefit of reliability growth modeling when very high levels of reliability are required [LS93].

Since empirical quantification of software failure rates is infeasible in the ultra-dependable region, we might consider the alternative approach of calculating the overall failure rate from those of smaller components. To be feasible, this approach must require relatively modest reliabilities of the components (otherwise we cannot measure them); the components must fail independently, or very nearly so (otherwise we do not achieve the multiplicative effect required to deliver ultra-dependability from components of lesser dependability), and the interrelationships among the components must be simple (otherwise we cannot use reliability of the components to calculate that of the whole). Ordinary software structures do not have this last property: the components communicate freely and share state, so one failure can corrupt the operation of other components [PvSK90]. However, specialized fault-tolerant system structures have been proposed that seek to avoid these difficulties.

One such approach is "multiple-version dissimilar software" [RTC92, Subsection 2.3.2] generally organized in the form of N-Version software [AL86, Avi85] or as "Recovery Blocks" [Ran75]. The idea here is to use two or more independently developed software versions in conjunction with comparison or voting to avoid system failures due to systematic failures in individual software versions. For the N-Version technique to be effective, failures of the separate software versions must be almost independent of each other.[10] The dif-

[9] Different reliability growth models often make very different predictions, and no single model is uniformly superior to the others; however, it is possible to determine which models are effective in a particular case, but only at modest reliability levels [BL92].

[10] For the Recovery Block technique to be effective, failure of the "Acceptance Test" must be almost independent of failures of the implementations comprising the body of the recovery block. The test and the body are intrinsically "more dissimilar" than N-Version components, which must all accomplish the same goal, but it is difficult to develop acceptance tests of the stringency required.

ficulty is that since independence cannot be assumed (experiments indicate that coincident failures of different versions are not negligible [ECK+91, KL86], and theoretical studies suggest that independent faults can produce correlated failures [EL85]—though the correlation can be negative [LM89]), the probability of coincident failures must be measured. But for this design approach to be effective, the incidence of coincident failures must be in the ultra-dependable region—and we are again faced with the infeasibility of experimental quantification of extremely rare events [BF93]. For these reasons, the degree of protection provided by software diversity "is not usually measurable" and dissimilar software versions do not provide a means for achieving safety-critical requirements, but "are usually used as a means of providing additional protection after the software verification process objectives for the software level... have been met" [RTC92, Subsection 2.3.2]. A further limitation on the utility of N-Version software is that the most serious faults are generally observed in complex functions such as redundancy management and distributed coordination. These employ fault-tolerant algorithms that work under specific fault-hypotheses. For example, fault-tolerant sensor-distribution algorithms are based on very carefully chosen voting techniques, and plausible alternatives can fail [LR93]. Supplying N implementations and additional voting does not necessarily make these functions more robust, but it certainly changes them and may violate the constraints and fault-hypotheses under which they work correctly. The daunting truth is that some of the core algorithms and architectural mechanisms in fault-tolerant systems are single points of failure: they just *have* to work correctly.

Another design technique suggested by the desire to achieve ultra-dependability through a combination of less-dependable components is use of unsynchronized channels with independent input sampling. Redundant computer channels (typically triplex or quadruplex) are required for fault tolerance with respect to random hardware failures in digital flight-control systems. The channels can operate either asynchronously or synchronously. One advantage claimed for the asynchronous approach is that the separate channels will sample sensors at slightly different times and thereby obtain slightly different values [McG90]. Thus, even if one channel suffers a systematic failure, the others, operating on slightly different input values, may not. Like design diversity, effectiveness of this "data diversity" depends on failures exhibiting truly random behavior: in this case the requirement is that activations of faults should not cluster together in the input space. As with independence in design diversity, experimental evidence suggests that this property cannot simply be assumed (there is some evidence for "error crystals" [DF90]) and it must therefore be measured. And as before, experimental determination of this property is infeasible at the low fault densities required.[11]

[11] Like N-Version software, asynchronous operation could also be proposed as a way to provide additional protection beyond that required and achieved by an individual software channel. This proposal overlooks the possibility that an asynchronous approach will complicate the overall design, having ramifications throughout the system, from fault detection, through reconfiguration, to the control laws. As the AFTI-F16 flight tests

If we cannot validate ultra-dependable software by experimental quantification of its failure rate, and we cannot make substantiated predictions about N-Version or other combinations of less-dependable software components, there seems no alternative but to base certification at least partly on other factors, such as analysis of the design and construction of the software, examination of the life-cycle processes used in its development, operational experience gained with similar systems, and perhaps the qualifications of its developers.

We might hope that if these "immanent" (i.e., mental) factors gave us a reasonable prior expectation of high dependability, then a comparatively modest run of failure-free tests would be sufficient to confirm ultra-dependability. Unfortunately, a Bayesian analysis shows that feasible time on test cannot confirm ultra-dependability, unless our prior belief is already one of ultra-dependability [LS93] (see also [MMN+92] for a detailed analysis of the probability of failure when testing reveals no failures). In other words, the requirement of ultra-dependability is so many orders of magnitude removed from the failure rates that can be experimentally determined in feasible time on test, that essentially *all* our assurance of ultra-dependability has to come from immanent factors such as examination of the life-cycle processes of its development, and review and analysis of the software itself.

We can distinguish two classes of immanent factors: *process* factors consider the methods used in the construction of the software, its documentation, the qualifications of the personnel, and so on, while *product* factors consider properties of the software system itself, such as the results of tests, and formal arguments that the software satisfies its requirements.

In current practice, most of the assurance for ultra-dependability derives from process factors. This is a rather chastening conclusion: assurance of ultra-dependability has to come from scrutiny of the software and scrupulous attention to the processes of its creation; since we cannot measure "how well we've done" we instead look at "how hard we tried." This, in essence, is the burden of DO-178B (and most other guidelines and standards for safety-critical software, e.g., [IEC86, MOD91a]). Of course, extensive testing is still required, but these tests are evaluated against criteria that measure how much of the system has been tested rather than whether they guarantee the presence or absence of certain properties. Thus, testing is perhaps best seen as serving to validate the assumptions that underpin the software design, and to corroborate the broad argument for its correctness, rather than as a validation of reliability claims.

Indeed, most standards for safety-critical software state explicitly that probabilities are not assigned or assessed for software that is certified by examination of its development processes:

> "...it is not feasible to assess the number or kinds of software errors, if any, that may remain after the completion of system design, development, and test" [FAA88, paragraph 7.i].

revealed [Mac88], this additional, unmastered complexity has become the primary source of failure in at least one system.

> "Development of software to a software level does not imply the assignment of a failure rate for that software. Thus, software levels or software reliability rates based on software levels cannot be used by the system safety assessment process as can hardware failure rates" [RTC92, Subsection 2.2.3].

(See also [MOD91b, paragraph 6.6 and Annex F].)

The infeasibility of experimental quantification of ultra-dependable software not only impacts the process of validating software, it also places constraints on the design of redundancy management mechanisms for tolerating hardware failures. Although the assumption of independent failures cannot be assumed for different software versions, it *is* a reasonable assumption for properly configured redundant hardware channels. Overall reliability of such a redundant system then depends on the failure rates of its components, and on properties of the architecture and implementation of the fault-tolerance mechanisms that tie it together (in particular, the coverage of its fault-detection mechanisms). The overall system reliability can be calculated using reliability models whose structure and transition probabilities are determined by hardware component reliabilities and by properties of the fault-tolerance mechanisms. These transition probabilities must either be calculated in some justifiable manner, or they must be measured: if they cannot be calculated or measured in feasible time on test, the architecture cannot be validated and its design must be revised.

This analysis motivates the methodology for fault-tolerant architectures known as "design for validation" [JB92], which is based on the following principles.

1. The system must be designed so that a complete and accurate reliability model can be constructed. All parameters of the model that cannot be deduced analytically must be measurable in feasible time under test.

2. The reliability model does not include transitions representing design faults; analytical arguments must be presented to show that design faults will not cause system failure.

3. Design tradeoffs are made in favor of designs that minimize the number of parameters that must be measured, and that simplify the analytic arguments.

Johnson and Butler [JB92] present a couple of representative examples to show how these principles might apply in practice.

Consider, for example, a dual-channel system (sometimes called a self-checking pair). Architectures of this type are widely used in full authority digital engine controls (FADECs) and autopilots. The idea is that each channel periodically checks its own health and also that of the other channel. When an error is detected, the afflicted channel either shuts itself down or, in some circumstances, it is shut down by the other channel. Now it is provably impossible for such an architecture always to make the correct diagnosis, so there will be circumstances where the chosen diagnosis algorithm will fail, and the wrong channel will be shut down. In order to calculate the overall reliability achieved, we therefore need to

know the failure rates of the individual processors (which can be looked up in reference works) and that of the diagnostic algorithm. Let us call the coverage (i.e., reliability) of this algorithm C. The only way to determine the value of C is by empirical tests. A Markov model allows the sensitivity of overall system reliability on the parameter C to be determined. Under plausible assumptions, it can be shown that $C > 0.9995$ will satisfy a system reliability goal of about 2×10^{-6} failures over a 10-hour mission (this example is from Johnson and Butler [JB92], where the details can be found). Further analysis reveals that 20,000 tests will be required to estimate the coverage of the diagnosis algorithm to the required level of statistical accuracy. If fault-injections can be performed at the rate of one per minute, then 333 hours (a little over 14 days) of continuous time on test will be required. This is feasible, and so this attribute of the proposed architecture can indeed be validated.

Now consider a different system, with a reliability requirement of $1 - 10^{-9}$ to be achieved using processors with failure rates of 10^{-5} per hour. This can be accomplished by a nonreconfigurable 5-plex architecture, provided the fault-masking algorithm has a coverage greater than 0.9999982. This time, analysis shows (the details are again in [JB92]) that over a million fault-injections will be required to validate satisfaction of this requirement. This amount of testing is infeasible (it is equivalent to 1.9 years on continuous test at one fault-injection per minute). We therefore have no alternative but to abandon this architecture in favor of one whose critical design parameters can be measured in feasible time—unless we can *prove* analytically that the chosen fault masking algorithm has coverage greater than 0.9999982. Now the usual approach to fault-tolerant design is Failure Modes and Effects Analysis (FMEA), which depends on enumerating all failure modes and designing a mechanism to counter each one. The problem here is not so much demonstrating the reliability of each countermeasure, but providing evidence (to the required level of statistical significance) that all failure modes have been accounted for. For a sufficiently simple system, it may be possible that such evidence could be provided analytically, but with a system as complex as a 5-plex of computers (with their associated sensors, cross-strapping, and voting) there is no alternative to experimental determination—and that will require an infeasible time on test.

But there is an alternative: it is possible to prove that certain architectures and algorithms can mask a single failure in a 5-plex, *no matter what the mode of failure* (these are based on Byzantine-resilient algorithms [LSP82, PSL80, Sch90]; they are proved correct without making any assumptions about the behavior of failed components). Thus in this case it is possible (indeed, necessary) to substitute mathematical analysis for (infeasible) experimental quantification of Markov transition probabilities.

To summarize this discussion: all software failures are of the systematic variety—there is nothing to go wrong but the processes of specification, design, and construction. Nonetheless, software failure can be treated as a random process and can be quantified probabilistically. However, validation of achieved failure rates by experimental quantification is infeasible in the ultra-dependable region. (This also places constraints on the design of fault-tolerant architectures, since system reliability models require accurately measured

or calculated probabilities of coverage for the redundancy-management and fault-tolerance software.) The realization that experimental validation is infeasible for software in the ultra-dependable region means that its validation must derive chiefly from analysis of the software and from control and evaluation of its development processes. Thus, the goals of the very disciplined life-cycle processes required by almost all standards and guidelines for safety-critical software are to minimize the opportunities for introduction of faults into a design, and to maximize the likelihood and timeliness of detection and removal of the faults that do creep in. The means for achieving these goals are structured development methods, extensive documentation tracing all requirements and design decisions, and careful reviews, analyses, and tests. The more critical a piece of software, the more stringent will be the application of these means of control and assurance.

Bibliography

[AL86] A. Avižienis and J. C. Laprie. Dependable computing: From concepts to design diversity. *Proceedings of the IEEE*, 74(5):629–638, May 1986.

[Avi85] Algirdas Avižienis. The *N*-Version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[BF93] Ricky W. Butler and George B. Finelli. The infeasibility of experimental quantification of life-critical software reliability. *IEEE Transactions on Software Engineering*, 19(1):3–12, January 1993.

[BL92] Sarah Brocklehurst and Bev Littlewood. New ways to get accurate reliability measures. *IEEE Software*, 9(4):34–42, July 1992.

[CDD90] Flaviu Cristian, Bob Dancey, and Jon Dehn. Fault-tolerance in the advanced automation system. In *Fault Tolerant Computing Symposium 20*, pages 6–17, Newcastle upon Tyne, UK, June 1990. IEEE Computer Society.

[CDM86] P. Allen Currit, Michael Dyer, and Harlan D. Mills. Certifying the reliability of software. *IEEE Transactions on Software Engineering*, SE-12(1):3–11, January 1986.

[DF90] Janet R. Dunham and George B. Finelli. Real-time software failure characterization. In *COMPASS '90 (Proceedings of the Fifth Annual Conference on Computer Assurance)*, pages 39–45, Gaithersburg, MD, June 1990. IEEE Washington Section.

[ECK+91] Dave E. Eckhardt, Alper K. Caglayan, John C. Knight, Larry D. Lee, David F. McAllister, Mladen A. Vouk, and John P. J. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE Transactions on Software Engineering*, 17(7):692–702, July 1991.

[EL85] Dave E. Eckhardt, Jr. and Larry D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering*, SE-11(12):1511–1517, December 1985.

[FAA88] Federal Aviation Administration. *System Design and Analysis*, June 21, 1988. Advisory Circular 25.1309-1A.

[FAA93] Federal Aviation Administration. *RTCA Inc., Document RTCA/DO-178B*, January 11, 1993. Advisory Circular 20-115B.

[FAA01] Federal Aviation Administration. *Technical Standard Order TSO-C153: Integrated Modular Avionics Hardware Elements*, December 17, 2001. Available for public comment (listed in the Federal Register on date shown).

[FTC95] IEEE Computer Society. *Fault Tolerant Computing Symposium 25: Highlights from 25 Years*, Pasadena, CA, June 1995. IEEE Computer Society.

[GAO92] United States General Accounting Office, Washington, DC. *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*, February 1992. GAO/IMTEC-92-26.

[Ham92] Dick Hamlet. Are we testing for true reliability? *IEEE Software*, 9(4):21–27, July 1992.

[Hec93] Herbert Hecht. Rare conditions: An important cause of failures. In *COMPASS '93 (Proceedings of the Eighth Annual Conference on Computer Assurance)*, pages 81–85, Gaithersburg, MD, June 1993. IEEE Washington Section.

[IEC86] International Electrotechnical Commission, Geneva, Switzerland. *Software for Computers in the Safety Systems of Nuclear Power Stations*, first edition, 1986. IEC Standard 880.

[JB92] Sally C. Johnson and Ricky W. Butler. Design for validation. *IEEE Aerospace and Electronic Systems Magazine*, 7(1):38–43, January 1992. Also in 10th AIAA/IEEE Digital Avionics Systems Conference, Los Angeles, CA, October 1991, pp. 487–492.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.

[KL86] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[Kop99] Hermann Kopetz. Elementary versus composite interfaces in distributed real-time systems. In *The Fourth International Symposium on Autonomous Decentralized Systems*, Tokyo, Japan, March 1999. IEEE Computer Society.

[KR93] Hermann Kopetz and Johannes Reisinger. The non-blocking write protocol NBW: A solution to a real-time synchronization problem. In *Real Time Systems Symposium*, pages 131–137, Raleigh-Durham, NC, December 1993. IEEE Computer Society.

[Lev86] Nancy G. Leveson. Software safety: Why, what and how. *ACM Computing Surveys*, 18(2):125–163, June 1986.

[LM89] B. Littlewood and D. R. Miller. Conceptual modeling of coincident failures in multiversion software. *IEEE Transactions on Software Engineering*, 15(12):1596–1614, December 1989.

[LR93] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23*, pages 402–411, Toulouse, France, June 1993. IEEE Computer Society. Reprinted in [FTC95, pp. 438–447].

[LS93] Bev Littlewood and Lorenzo Strigini. Validation of ultrahigh dependability for software-based systems. *Communications of the ACM*, pages 69–80, November 1993.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[LT82] E. Lloyd and W. Tye. *Systematic Safety: Safety Assessment of Aircraft Systems*. Civil Aviation Authority, London, England, 1982. Reprinted 1992.

[Mac88] Dale A. Mackall. Development and flight test experiences with a flight-crucial digital control system. NASA Technical Paper 2857, NASA Ames Research Center, Dryden Flight Research Facility, Edwards, CA, 1988.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.

[McG90] John G. McGough. The Byzantine Generals Problem in flight control systems. In *AIAA Second International Aerospace Planes Conference*, Orlando, FL, October 1990. AIAA paper 90-5210.

[McM99] K. L. McMillan. Circular compositional reasoning about liveness. In Laurence Pierre and Thomas Kropf, editors, *Advances in Hardware Design and Verification: IFIP WG10.5 International Conference on Correct Hardware Design and Verification Methods (CHARME '99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 342–345, Bad Herrenalb, Germany, September 1999. Springer-Verlag.

[MIO87] John D. Musa, Anthony Iannino, and Kazuhira Okumoto. *Software Reliability—Measurement, Prediction, Application*. McGraw Hill, New York, NY, 1987.

[MMN+92] Keith W. Miller, Larry J. Morell, Robert E. Noonan, Stephen K. Park, David M. Nicol, Branson W. Murrill, and Jeffrey M. Voas. Estimating the probability of failure when testing reveals no failures. *IEEE Transactions on Software Engineering*, 18(1):33–43, January 1992.

[MOD91a] UK Ministry of Defence. *Interim Defence Standard 00-55: The Procurement of Safety Critical Software in Defence Equipment*, April 1991. Part 1, Issue 1: Requirements; Part 2, Issue 1: Guidance.

[MOD91b] UK Ministry of Defence. *Interim Defence Standard 00-56: Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment*, April 1991.

[NT00] Kedar S. Namjoshi and Richard J. Trefler. On the completeness of compositional reasoning. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification, CAV '2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 139–153, Chicago, IL, July 2000. Springer-Verlag.

[Per84] Charles Perrow. *Normal Accidents: Living with High Risk Technologies*. Basic Books, New York, NY, 1984.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[PvSK90] David L. Parnas, A. John van Schouwen, and Shu Po Kwan. Evaluation of safety-critical software. *Communications of the ACM*, 33(6):636–648, June 1990.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[RTC92] Requirements and Technical Concepts for Aviation, Washington, DC. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, December 1992. This document is known as EUROCAE ED-12B in Europe.

[Rus99] John Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Available at `http://www.csl.sri.com/~rushby/abstracts/partitioning`,

and http://techreports.larc.nasa.gov/ltrs/PDF/1999/cr/NASA-99-cr209347.pdf; also issued by the FAA.

[Rus01] John Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, September 2001. Available at http://www.csl.sri.com/~rushby/abstracts/buscompare.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[Sim90] H. R. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings, Part E: Computers and Digital Techniques*, 137(1):17–30, January 1990.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
12-2002	Contractor Report	

4. TITLE AND SUBTITLE
Modular Certifi cation

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
John Rushby

5d. PROJECT NUMBER
NCC1-377

5e. TASK NUMBER

5f. WORK UNIT NUMBER
728-30-10-01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
SRI International
333 Ravenswood Ave
Menlo Park, California 94025

8. PERFORMING ORGANIZATION REPORT NUMBER
Project 11003, Deliverable 4

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/CR–2002–212130

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassifi ed-Unlimited
Subject Category 62
Availability: NASA CASI (301) 621-0390 Distribution: Standard

13. SUPPLEMENTARY NOTES
An electronic version can be found at http://techreports.larc.nasa.gov/ltrs/ or http://techreports.larc.nasa.gov/cgi-bin/NTRS.
Langley Technical Monitor: Paul S. Miner. Report prepared by SRI International under NASA Cooperative agreement NCC1-377 with Honeywell International

14. ABSTRACT
Airplanes are certifi ed as a whole: there is no established basis for separately certifying some components, particularly software-intensive ones, independently of their specifi c application in a given airplane. The absence of separate certifi cation inhibits the development of modular components that could be largely "precertifi ed" and used in several different contexts within a single airplane, or across many different airplanes.

In this report, we examine the issues in modular certifi cation of software components and propose an approach based on assume-guarantee reasoning. We extend the method from verifi cation to certifi cation by considering behavior in the presence of failures. This exposes the need for partitioning, and separation of assumptions and guarantees into normal and abnormal cases. We then identify three classes of property that must be verifi ed within this framework: safe function, true guarantees, and controlled failure.

15. SUBJECT TERMS
Certifi cation, Integrated Modular Avionics

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
Unclassifi ed	Unclassifi ed	Unclassifi ed	Unlimited	48	19b. TELEPHONE NUMBER *(Include area code)* (301) 621-0390