SRI International

CSL Technical Report ● September 2001
Minor revision June 2002

Modular Certification

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

This research was supported by NASA Langley Research Center under contract NAS1-20334 and Cooperative Agreement NCC-1-377 with Honeywell Tucson, and by the DARPA MoBIES program under contract F33615-00-C-1700 with US Air Force Research Laboratory.

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.

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.

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.

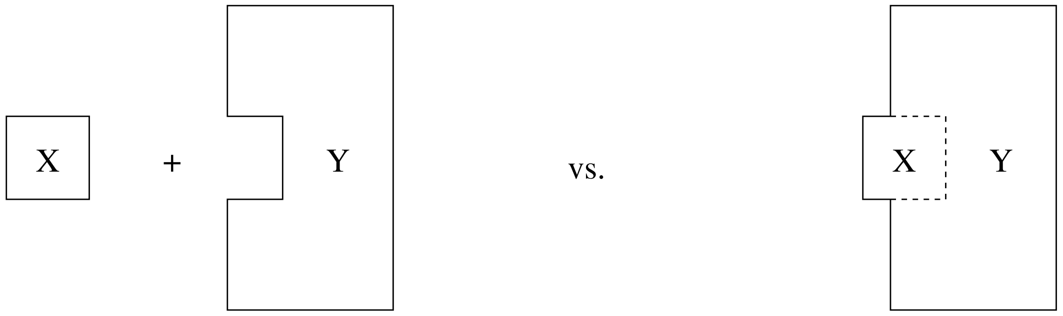

Figure 2.1: Modular vs. Traditional Certification

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.

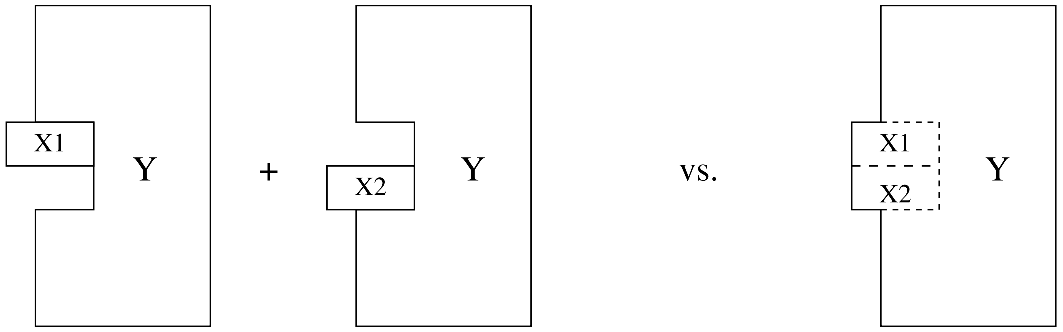

Figure 2.2: Reinterpretation of Modular Certification

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.

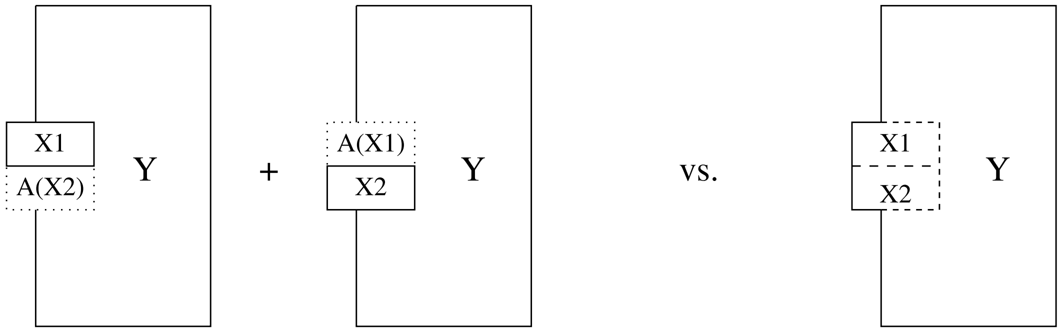

Figure 2.3: Assume-Guarantee Modular Certification

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 computations, 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 [Rus01a]. 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 [Rus01a].

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.

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.

Chapter 3

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. McMillan [McM99] presents a sound rule that seems suitable for the applicaitons considered here. Its soundness has been formally verified in PVS [Rus01b].

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.

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.

Bibliography

[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). 1

[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. 9

[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. 9

[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. 11

[Per84] Charles Perrow. *Normal Accidents: Living with High Risk Technologies*. Basic Books, New York, NY, 1984. 12

[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. 7

[Rus01a] 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`. 7

[Rus01b] John Rushby. Formal verification of McMillan's compositional assume-guarantee rule. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, September 2001. 11

[Sim90] H. R. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings, Part E: Computers and Digital Techniques*, 137(1):17–30, January 1990. 9