Invited paper; to appear in proceedings of FACS'11 (Springer LNCS).

Composing Safe Systems*

John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025 USA

Abstract. Failures in component-based systems are generally due to unintended or incorrect interactions among the components. For safety-critical systems, we may attempt to eliminate unintended interactions, and to verify correctness of those that are intended. We describe the value of partitioning in eliminating unintended interactions, and of assumption synthesis in developing a robust foundation for verification. We show how model checking of very abstract designs can provide mechanized assistance in human-guided assumption synthesis.

1 Introduction

We build systems from components, but what makes something a *system* is that its properties and behaviors are distinct from those of its components. As engineers and designers, we wish to predict and calculate the properties of systems from those of their interconnected components, and we are quite successful at doing this, *most of the time*. For many systems and properties, "most of the time" is good enough: we can live with it if our laptop occasionally locks up, our car doesn't start, or our music player seems to lose our playlists. But we will be considerably more aggrieved if our laptop catches fire, our car fails to stop, or our music player loses the songs that we purchased. As we move from personal systems to those with wider impact and from properties about normal function to those that concern safety or security, so "most of the time" becomes inadequate: we want those properties to be true *all the time*.

Often, properties that we want to be true "all the time" fail to be so, and subsequent investigations generally reveal some unexpected interaction among the system's components. Thus, attempts to reason about the properties of systems by combining or *composing* the properties of their components, while generally successful for "most of the time" properties, are less successful for "all the time" properties. It is for this reason that regulatory bodies examine only complete systems (e.g., the FAA certifies only airplanes and engines) and not components: they need to consider the behaviors and possible interactions of multiple components in the context of a specific system.

* This work was supported by National Science Foundation Grant CNS-0720908. The content is solely the responsibility of the author and does not necessarily represent the official views of NSF.

Now, although it is generally infeasible at present to guarantee critical "all the time" properties by compositional (or "modular") methods, it is a good research topic to investigate why this is so, and how we might extend the boundaries of what is feasible in this area. Safety, in the sense of causing no harm to the public, is one of the most demanding properties, and so the motivation for the title of this paper is to indicate a research agenda focused on methods that might allow certification of safety for complex systems by compositional means.

As mentioned, system safety failures and the attendant flaws in compositional reasoning are generally due to unanticipated interactions among components. These interactions can be classified into those that exploit a previously unanticipated pathway for interaction, and those that are due to unanticipated behavior along a known interaction pathway. One way to control the first class of unanticipated interactions is to use integration frameworks that restrict the pathways available for component interactions; in avionics, this approach is called "partitioning" and it is the topic of Section 2.

There are two complementary ways to deal with the second class of unanticipated interactions: one is to augment components with wrappers, monitors, or other mechanisms that attempt to limit interactions to those that are needed to accomplish the purpose for which the interaction pathway was established; the other is to actually verify correctness of some interactions. Ideally, the verification should be done compositionally: that is, we verify properties of components considered separately, then from those properties we derive properties of their interaction. The verification of each component has to consider the environment in which it will operate, and that environment is composed of the other components with which it interacts. This seems contrary to a pure conception of component-based design, because it looks as if each component needs to consider the others during its design. One way to avoid this is to calculate the weakest environment in which a component can perform its task; this is among the topics of Section 3, which mainly focuses on assume/guarantee and methods for assumption synthesis. Brief conclusions are provided in Section 4.

2 Partitioning

Aircraft are safe, yet employ many interacting subsystems, so the techniques they employ are worthy of interest. Traditionally, the various avionics functions on board aircraft were provided by fairly independent subsystems loosely integrated as a "federated" system. This meant that the autopilot, for example, had its own computers, replicated for redundancy, and so did the flight management system. The two system would communicate though the exchange of messages, but their relative isolation provided a natural barrier to the propagation of faults: a faulty autopilot might send bad data to the flight management system, but could not destroy its ability to calculate or communicate.

Modern aircraft employ Integrated Modular Avionics (IMA) where many critical functions share the same computer system and communications networks, and so there is naturally concern that a fault in one function could

propagate to others sharing the same resources. Hence, the notion of "robust partitioning" has developed [1]: the idea is that applications that use the shared resources should be protected from each other as if they were not sharing and each had their own private resources.

The primary resources that require partitioning are communication and computation: i.e., networks and processors. For networks, the concern is that a faulty or malicious component will not adhere to the protocol and will transmit over other traffic, or will transmit constantly, thereby denying good components the ability to communicate. The only way to provide partitioning in the face of these threats is to employ redundancy, so that components' access to the network is mediated by additional components that limit the rate or the times at which communication can occur. Of course, these additional components and the mechanisms they employ may themselves be afflicted by faults (e.g., transient hardware upsets caused by ambient radiation), and so the design and assurance of these partitioning network technologies are very demanding [2], but they are now reasonably well understood and available "off the shelf."

For processors, the concerns are that faulty or malicious processes will write into the memory of other processes, monopolize the CPUs, or corrupt the processor's state. Partitioning against these threats can be provided by a strong operating system or, more credibly, by a hypervisor or virtual machine environment; minimal hypervisors specialized to the partitioning function are known as "separation kernels" [3] and, like partitioning networks, efficient and highly assured examples are now available "off the shelf."

Partitioning for the basic resources of communication and computation can be leveraged to provide partitioning for additional resources synthesized above them, such as file systems. A collection of partitioning resources may be configured to specify quite precisely what software components are allocated to each partition and what interactions are allowed with other components (the configuration for an IMA is many megabytes of data). Such configurations, which may be portrayed as box and arrow diagrams and formalized as "policy architectures" [4], eliminate undesired paths for interaction and provide direct assurance for certain system-level properties, such as some notions of security.

Fig. 1. A Partitioned System Configured to Support the System Purpose

The properties for which partitioning, on its own, provides adequate enforcement and assurance are those that can be stated in terms of the absence of information flow. As mentioned, certain security concerns are of this kind (e.g., "no flow from secret to unclassified"), but most properties also concern the computations that take place in (some of) the partitions. For example, many

secure systems *do* allow flow from secret to unclassified provided the information concerned is suitably "sanitized" by some function interposed in the flow, as portrayed by the minimal policy architecture of Figure 1. The partitioning configuration ensures the sanitizer cannot be bypassed, but we still require assurance that the sanitizer does its job. More complex properties, such as most notions of safety, cannot be ensured by individual components; instead, they emerge from the interactions of many—but partitioning eliminates unintended interactions and allows us to focus on correctness of the intended interactions, which is the topic of the next section.

3 Assumption Synthesis

If we suppose that "traditional software engineering" is able to develop systems that work "most of the time" then it might be possible to turn these into systems that work "all the time" by simply blocking untanticipated events and interactions that might lead to failure, or by controlling the propagation of any failures that are precipitated. These topics are addressed by a variety of techniques such as systematic exception handling [5], anomaly detection [6], safety kernels [7] and enforceable security [8], and runtime monitoring [9]. All these techniques merely reduce the frequency or severity of failures (e.g., by turning "malfunction" or "unintended function" into "loss of function") rather than eliminate them. However, they can be very valuable in systems with many layers of redundancy or fault management, since these often cope very well with the "clean" failure of subsystems, but less well with their misbehavior. Some aircraft systems employ "monitored architectures," where a very simple component monitors the system safety property and shuts down the operational component if this is violated; these architectures can support rather strong assessments of reliability [10].

To get from clean failures to true "all the time" systems by compositional means, we need to be able to calculate the properties of the composed system from those of its components; if the calculation is automated, then it can support an iterative design loop: if the composed system does not satisfy the properties required, then we modify some of the components and their properties and repeat the calculation.

The established way to calculate the properties of interacting components is by assume/guarantee reasoning [11]: we verify that component A delivers (or guarantees) property p on the assumption that its environment delivers property q, and we also verify that B guarantees q assuming p; then when A and B are composed, each becoming the environment of the other, we may conclude (under various technical conditions) that their composition $A||B$ guarantees both p and q. There is, however, a practical difficulty with this approach: if A and B are intended for general use, they are presumably developed in ignorance of each other, and it will require good fortune or unusual prescience that they should each make just the right assumptions and guarantees that they fit together perfectly.

Shankar proposes an alternative approach [12] that treats assumptions as abstract components; here, we establish that p is a property of A in the context of an ideal environment E; if we can then show that B as a refinement of E, then the composition of A and B also delivers p. This requires less prescience because we do not need to know about B at the time we design A; we do, however, need to postulate a suitable E.

One interesting idea is to design A, then *calculate* E as the weakest environment under which we can guarantee that A delivers p. When A is a concrete state machine, this can be done using L^* learning [13]. Early in the design process, however, we are unlikely to have developed A to the point where it is available as a fully concrete design; in this case we can often perform assumption synthesis interactively using infinite bounded model checking (inf-BMC)

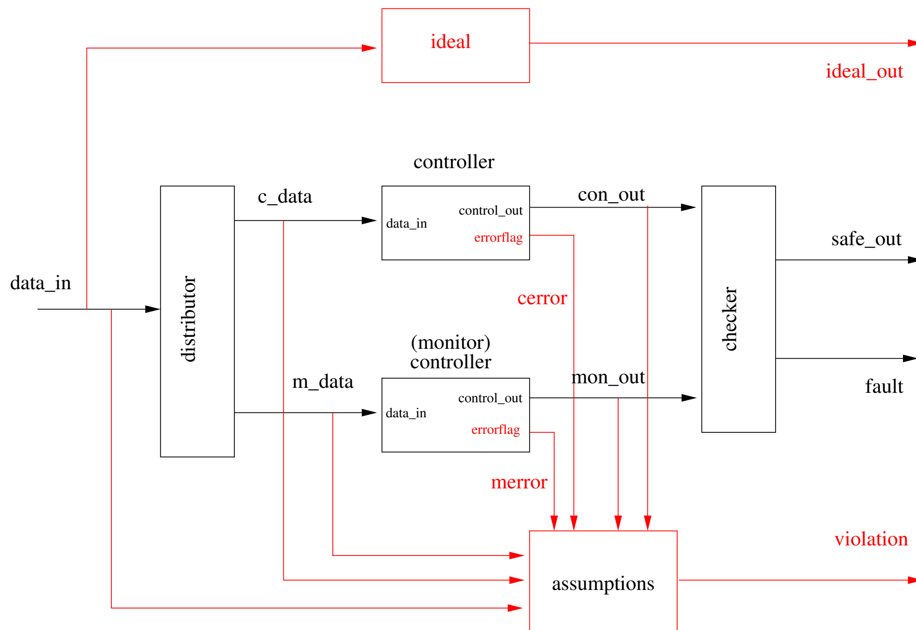

Fig. 2. A Self-Checking Pair, and Additional Components Used in Analysis

Inf-BMC performs bounded model checking on state machines defined over the theories supported by an SMT solver (i.e., a solver for *Satisfiability Modulo Theories*) [14]; these theories include equality over uninterpreted functions, possibly constrained by axioms, so it is possible to specify very abstract state machines. An example, taken from [15], is illustrated in Figure 2. Here, the goal is to deduce the assumptions under which a self-checking pair works correctly.

Self-checking pairs are used quite widely in safety-critical systems to provide protection against random hardware faults: two identical processors perform the

same calculations and their results are compared; if they differ the pair shuts down (thereby becoming a "fail-stop" processor [16]) and some higher-level fault management activity takes over. Obviously, this does not work if both processors become faulty and compute the *same* wrong result. We would like to learn if there are any other scenarios that can cause a self-checking pair to deliver the wrong result; we can then assess their likelihood (for example, the double fault scenario just described may be considered extremely improbable) and calculate the overall reliability of this architecture.

The scenarios we wish to discover are, on one hand, the *hazards* to the design and, on the other (i.e., when negated), its assumptions. At the system level, hazard analysis is the cornerstone of safety engineering [17]; in component-based design, assumption discovery—its dual—could play a similar rôle: it helps us understand when it is safe for one component to become the environment for another.

Because of its context (the environment of a self-checking pair really is the natural environment, rather than another system), this example is closer to hazard discovery than assumption synthesis—but since these are two sides of the same coin, it serves to illustrate the technique. The idea is that the `controller` and the `monitor` are identical fault-prone processors that compute some uninterpreted function $f(x)$; a `distributor` provides copies of the input x to both processors and the results are sent to a `checker`; if the results agree, the checker passes one of them on as `safe_out`, otherwise it raises a `fault` flag. The `distributor` as well as the two processors can deliver incorrect outputs, but for simplicity of exposition the checker is assumed to be perfect (the checker can be eliminated by having the `controller` and `monitor` cross-check their results). An `ideal` processor, identical to the others but not subject to failures, serves as the correctness oracle, and an `assumptions` module, which operates as a *synchronous observer*, encodes the evolving assumptions. In the figure, the `ideal` and `assumptions` modules and their associated data are shown in red to emphasize that these are artifacts of analysis, not part of the component under design.

Initially, the assumptions are empty and we use inf-BMC to probe correctness of the design (i.e., we attempt to verify the claim that if the `fault` flag is down, then `safe_out` equals `ideal_out`). We obtain a counterexample that alerts us to a missing assumption; we add this assumption and iterate. The exercise terminates after the following assumptions are discovered.

1. When both members of the self-checking pair are faulty, their outputs should differ (this is the case we already thought of).
2. When the members of the pair receive different inputs[1] (i.e., when the distributor is faulty), their outputs should differ. There are two subcases here.

[1] Readers may wonder how a distributor, whose implementation could be as simple as a solder joint connecting two wires, can alter the values it delivers to the processors; one possibility is it adds resistance and drops the voltage: one processor may see a weak voltage as a 1, the other as a 0.

(a) Neither member of the pair is faulty. The scenario here is that instead of sending the correct value x, the distributor sends y to one member of the pair and z to another, but $f(y) = f(z)$ (and $f(y) \neq f(x)$).
(b) One or both of the pair are faulty. Here, the scenario is the distributor sends the correct value x to the faulty member, and an incorrect value y to the nonfaulty member, but $f(y) = f'(x)$, where f' is the computation of the faulty member.

3. When both members of the pair receive the same input, it is the correct input (i.e., the distributor should not create a bad value and send it to both members of the pair).

Inf-BMC can verify that the self-checking pair works, given these four assumptions, so our next task is to examine them.

Cases 1 and 2(b) require double faults and may be considered improbable for that reason. Case 2(a) is interesting because it probably would not be discovered by finite state model checking, where we do not have uninterpreted functions: instead, the usual way to analyze an "abstract" design is to provide a very simple "concretization," such as replacing $f(x)$ by $x + 1$, which masks any opportunity to find the fault. This case is also interesting because, once discovered, it can be eliminated by modifying the design: simply cause each member of the pair to pass its input as well as its output to the checker; since both processors are nonfaulty, the inputs will be passed correctly to the checker, which will then raise the fault flag because it sees that the inputs differ. That leaves case 3 as the one requiring further consideration (which we do not pursue here) by those who would use a self-checking pair.

This example has illustrated, I hope, how automated methods such as inf-BMC can be used to help calculate the weakest assumptions required by a component, and thereby support the design of systems in which components' assumes and guarantees are mutually supportive, without requiring prescience.

4 Conclusions

All fields of engineering build on components, and it is natural that computer science should do the same. However, component-based systems can be rather more challenging in computer science than in other fields because of the complexity of interaction—unintended as well as in intended—that is possible. This complexity of interaction becomes even more vexatious when we aim to develop safety-critical and other kinds of system that must work correctly all the time. (Perrow [18] argues that unintended interactions and their enablers, "interactive complexity" and "tight coupling," are the primary causes of disasters in all engineering fields; however, computer systems generally have more complexity of these kinds, even in normal operation, than those of other fields.)

A plausible way to develop safe systems from components begins by eliminating unintended interactions, then ensures that the intended interactions are correct.

Unintended interactions can be divided into those that deliver unintended behavior along intended pathways, and those that employ unintended pathways. I have outlined techniques that can ameliorate these concerns. *Partitioning* aims to eliminate unintended pathways for interaction in networks and processors and higher-level resources built on these. Partitioning guarantees "preservation of prior properties" when new components are added to an existing system; it also seems sufficient, on its own, to guarantee certain kinds of information flow security properties, and to simplify the assured construction of more complex properties of this kind [19].

With unintended pathways controlled by partitioning, we can turn to interactions along known pathways; we need to ensure that unintended interactions are eliminated and the intended ones are correct. Various techniques related to wrapping and monitoring can be used to control faulty inputs and outputs; suitable wrappers or monitors can often be generated automatically by formal synthesis from component assumptions and high-level system requirements.

These techniques can eliminate, or at least reduce, the incidence of unintended and faulty interactions, but ultimately we need to calculate the composed behaviors of interacting components and verify their correctness. Traditional methods for compositional verification by assume/guarantee reasoning demand a degree of prescience to ensure that the assumes of one component are met by the guarantees of another that was designed in ignorance of it. One way to lessen this need for prescience is to derive the weakest assumptions under which a component can deliver its guarantees, and I sketched how inf-BMC can be used to provide automated assistance in this process (which is closely related to hazard analysis) very early in the design cycle.

Compositional design and assurance for critical systems that must function correctly, or at least safely, all the time, are challenging and attractive research topics. Further systematic examination and study of the methods and directions I described could be worthwhile, but fresh thinking would also be welcome.

References

1. Requirements and Technical Concepts for Aviation Washington, DC: DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations. (2005) Also issued as EUROCAE ED-124 (2007). 3
2. Rushby, J.: Bus architectures for safety-critical embedded systems. In Henzinger, T., Kirsch, C., eds.: EMSOFT 2001: Proceedings of the First Workshop on Embedded Software. Volume 2211 of Lecture Notes in Computer Science., Lake Tahoe, CA, Springer-Verlag (2001) 306–323 3
3. Rushby, J.: The design and verification of secure systems. In: Eighth ACM Symposium on Operating System Principles, Asilomar, CA (1981) 12–21 (ACM *Operating Systems Review*, Vol. 15, No. 5). 3
4. Boettcher, C., DeLong, R., Rushby, J., Sifre, W.: The MILS component integration approach to secure information sharing. In: 27th AIAA/IEEE Digital Avionics Systems Conference, St. Paul, MN, The Institute of Electrical and Electronics Engineers (2008) 3

5. Cristian, F.: Exception handling and software fault tolerance. IEEE Transactions on Computers **C-31** (1982) 531–540 4
6. Chandola, V., Banerjee, A., Kumar, V.: Anomaly detection: A survey. ACM Computing Surveys **41** (2009) 4
7. Rushby, J.: Kernels for safety? In Anderson, T., ed.: Safe and Secure Computing Systems. Blackwell Scientific Publications (1989) 210–220 4
8. Schneider, F.: Enforceable security policies. ACM Transactions on Information and System Security **3** (2000) 30–50 4
9. Havelund, K.: Program Monitoring; Course material for part II of Caltech CS 119. (May) Available at `http://www.runtime-verification.org/course/`. 4
10. Littlewood, B., Rushby, J.: Reasoning about the reliability of fault-tolerant systems in which one component is "possibly perfect". IEEE Transactions on Software Engineering (2011) Accepted for publication. 4
11. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Transactions on Programming Languages and Systems **5** (1983) 596–619 4
12. Shankar, N.: Lazy compositional verification. In de Roever, W.P., Langmaack, H., Pnueli, A., eds.: Compositionality: The Significant Difference (Revised lectures from International Symposium COMPOS'97). Volume 1536 of Lecture Notes in Computer Science., Bad Malente, Germany, Springer-Verlag (1997) 541–564 5
13. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Component verification with automatically generated assumptions. International Journal on Automated Software Engineering **12** (2005) 297–320 5
14. Rushby, J.: Harnessing disruptive innovation in formal verification. In Hung, D.V., Pandya, P., eds.: Fourth International Conference on Software Engineering and Formal Methods (SEFM), Pune, India, IEEE Computer Society (2006) 21–28 5
15. Rushby, J.: A safety-case approach for certifying adaptive systems. In: AIAA Infotech@Aerospace Conference, Seattle, WA, American Institute of Aeronautics and Astronautics (2009) AIAA paper 2009-1992. 5
16. Schlichting, R.D., Schneider, F.B.: Fail-stop processors: An approach to designing fault-tolerant computing systems. ACM Transactions on Computer Systems **1** (1983) 222–238 6
17. Leveson, N.G.: Safeware: System Safety and Computers. Addison-Wesley (1995) 6
18. Perrow, C.: Normal Accidents: Living with High Risk Technologies. Basic Books, New York, NY (1984) 7
19. Chong, S., van der Meyden, R.: Using architecture to reason about information security. Technical report, University of New South Wales (2009) 8