Invited talk for 8th International Symposium on Formal Aspects of Component Software, FACS'11, Oslo Norway, 14–16 Sept 2011

Composing Safe Systems

John Rushby

Computer Science Laboratory
SRI International
Menlo Park, CA

Introduction

- We build systems from components
- But what makes something a system is that its properties are distinct from those of its components
  - New properties emerge from component interactions
- However, we can generally calculate and predict the system behavior from those of its components and their interconnection
- This is what engineering is all about, and it works pretty well, most of the time
- And for many systems and properties, this is good enough
- But for certain kinds of systems and properties (quintessentially, safety-critical ones), it is insufficient
- We need properties to be true all of the time

Failures

- When a system fails, investigation often reveals unexpected interactions among components
  - One component does something unexpected
    (e.g., fails non-silently)
  - Other components react badly
  - The world falls apart
- It is for this reason that the FAA, for example, does not certify components, only complete airplanes and engines
  - They need to consider the possible interactions of multiple components in the context of a specific system
  - Components seldom advertise their failures; in a specific system, can focus on the hazards posed by each

A Research Agenda

- It is currently infeasible to guarantee critical all the time properties by compositional or modular reasoning
- Have to look at specific systems (like the FAA)
- But it is a good research topic to figure out why this is so and what can be done about it
- Safety, in the sense of causing no harm to the public, is one of the most demanding properties
- So the motivation for my title is to indicate a research agenda focused on methods that might allow certification of safety for complex systems by compositional means

Two Kinds of Unanticipated Interactions

- Those that exploit a previously unanticipated pathway for interaction
  - Can be controlled by partitioning
- Those due to unanticipated behavior along a known pathway
  - Can be controlled by monitoring, wrapping, etc., and by anticipating the unanticipated
- I'll sketch these, and focus on the last

Partitioning

- Aircraft employ many interacting subsystems, yet are safe
- Traditionally, they used a federated architecture
  - Each subsystem (autopilot, brakes, yaw damper etc.) had its own computer system
  - Often replicated for reliability
  - Separate subsystems could communicate through exchange of messages
  - But their relative isolation provided a natural barrier to fault propagation
- Modern aircraft use Integrated Modular Avionics (IMA)
  - Subsystems share resources
  - Partitioning restores same fault isolation as federated system

Partitioning Mechanisms

- Partitioning for processors is achieved by a minimized OS kernel/hypervisor (a separation kernel)
- Partitioning for networks requires special engineering to limit disruption due to faulty (e.g., babbling) nodes
  - Control either rate, or time of access
    (cf. AFDX, TTA/TTE)
- Together, these guarantee information flows specified by box and arrow diagrams

Why Partitioning?

Why do I need partitioning when my stuff is formally verified and is correct?

1. Your stuff may be correct, but the other guy's might not be
2. Even your stuff is subject to random hardware faults
   (SEUs, HIRF etc)

Partitioning guarantees preservation of prior properties

Sometimes, Partitioning Is All You Need

- Recall, partitioning guarantees information flows specified by box and arrow diagrams (a policy architecture)
- And sometimes this is all you need
- Certain security properties are like this

- Sometimes you need some of the nodes to guarantee certain properties (like the sanitizer above)
- Exercise: formalize this
  - Cf. MILS, and recent work by Ron Van Der Meyden
  - It depends on intransitive noninterference

Related Techniques

- There are several related ideas in this space
- Safety kernels, enforceable security, anomaly detection, wrapping, runtime monitoring, etc.
- Very simple monitors may be possibly perfect
- The reliability of a monitored system is (roughly) the product of the reliability of the primary system and the probability of perfection of the monitor
- See a forthcoming TSE paper by Bev Littlewood and me

From Controlling The Bad To Making Good

- Looked at methods that stop components doing bad things
- Now look at how to ensure that components do good things

Classical Compositional Reasoning

- Typically assume/guarantee
- Roughly, verify that component A delivers (or guarantees) property p, on the assumption its environment guarantees q
- And that component B guarantees property q, on the assumption its environment guarantees p
- When these are composed, each becomes the environment of the other and their composition $A||B$ guarantees $p \wedge q$
- But if these are true components, each is surely designed in ignorance of the other, so it requires prescience (or good fortune) that they each assume and guarantee just the right properties to match up

Lazy Compositional Reasoning

- Shankar has an alternative lazy approach
- Establish that A delivers p in the context of an ideal environment E
- Later need to show that B refines E
- Less prescience needed: don't need to know about B when we design A
- But we do need to postulate a suitable E

Assumption Synthesis

- An alternative is to design A, then calculate or synthesize the weakest environment under which it guarantees p
- When A is a concrete state machine, can do this by L^* learning
- But early in the lifecycle, we have only a sketch for A
- Want to calculate the assumptions needed to make to work
- If these are implausible, revise the design
- If reasonable, note them as the properties that must be guaranteed by its environment when used in a system

Assumptions and Hazards

- In safety-critical systems, circumstances that could lead to safety failure are called hazards
- Safety-critical engineering is about finding all the hazards, and showing that each is countered (eliminated or mitigated) effectively
- So assumption synthesis is related to hazard discovery
  - They are duals

Assumption Discovery Using Inf-BMC

- Inf-BMC does bounded model checking (BMC) on state machines defined over theories supported by an SMT solver
  - SMT is Satisfiability Modulo Theories
  - Roughly, combines SAT solving with decision procedures for theories like equality with uninterpreted functions, linear arithmetic, etc.
  - The biggest advance in formal methods in last 20 years
  - Performance honed by annual competition
- State space is potentially infinite, hence inf-BMC
- Combines the expressiveness and abstractness of theorem proving with the automation of model checking
- Highly abstract components can be specified using uninterpreted functions, possibly constrained by axioms

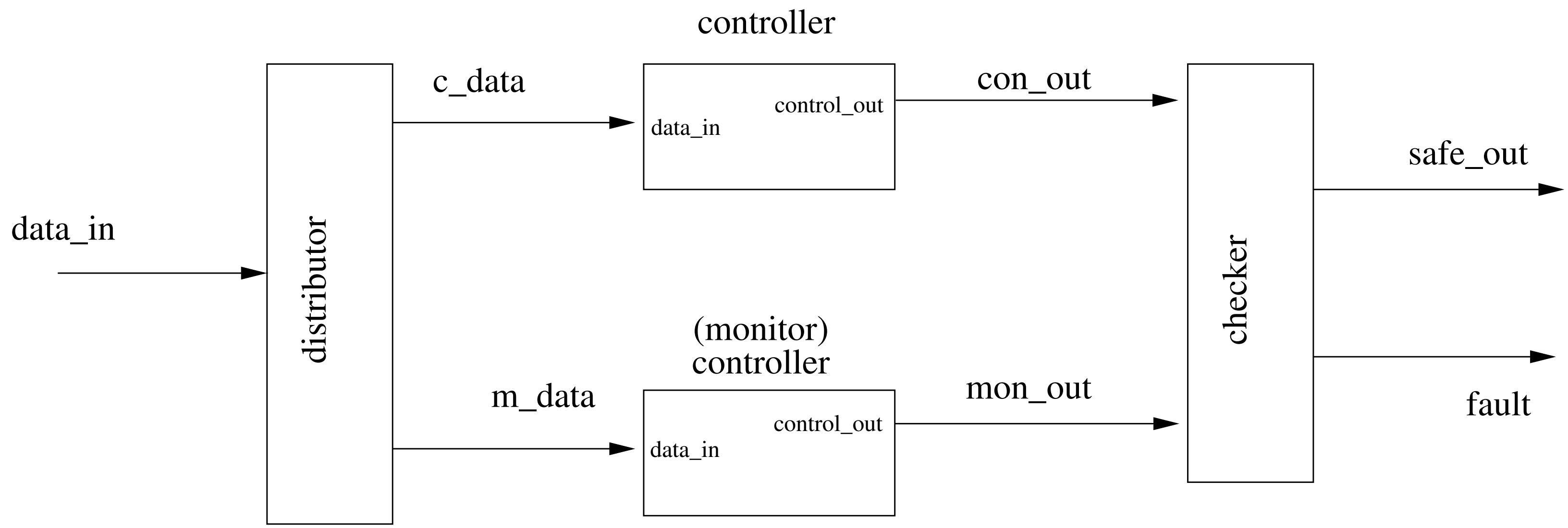
Example: Protecting Against Random Faults

- Components that fail by stopping cleanly are fairly easy to deal with
- The danger is components that do the wrong thing
- We have to eliminate design faults by analysis (that's what we're doing here), but we still have to worry about random faults
  - When an α-particle flips a bit in your instruction counter
- Our goal is to design a component that fails cleanly in the presence of random faults

Example: Self-Checking Pair

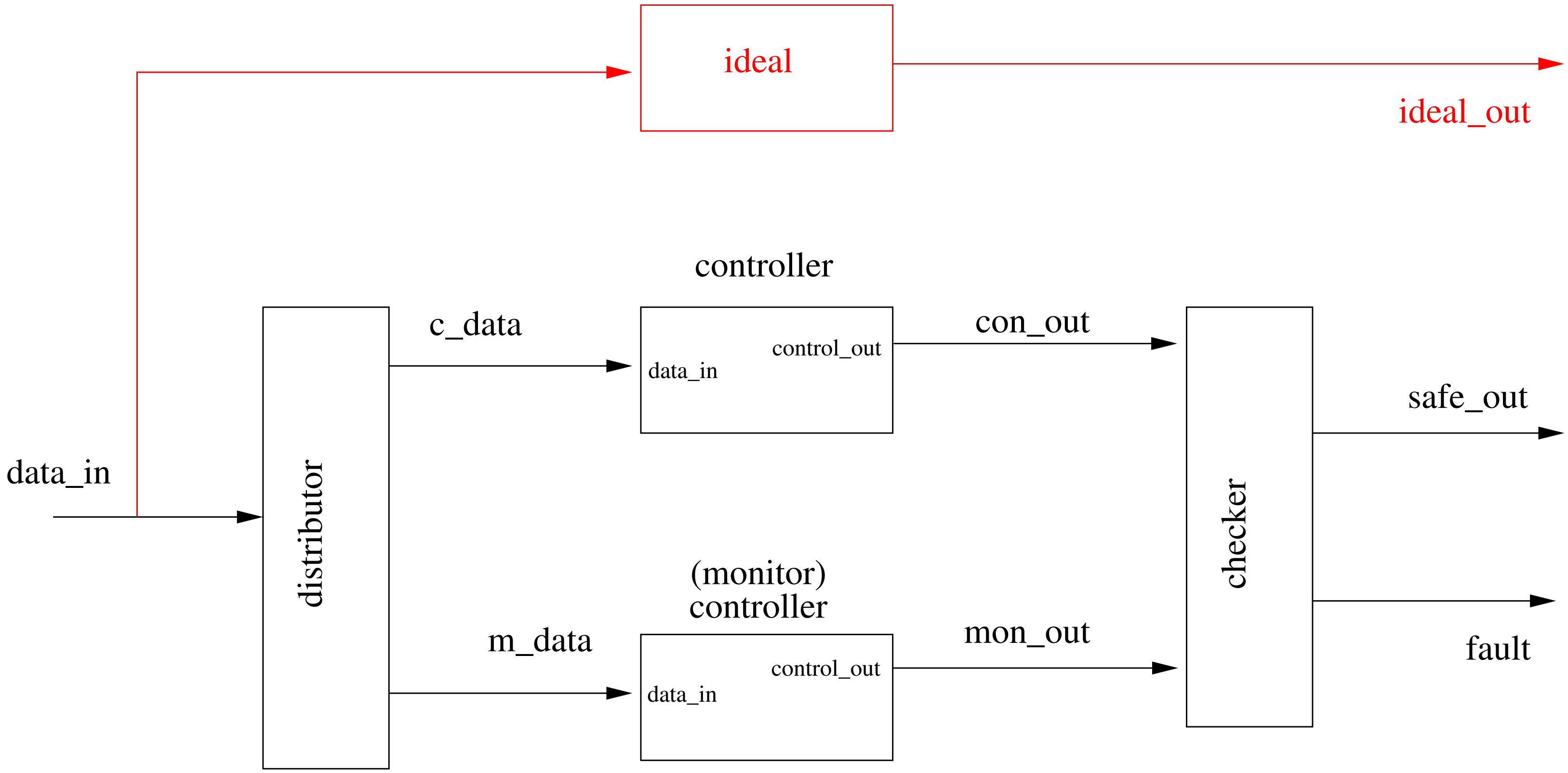
- If they are truly random, faults in separate components should be independent
  - Provided they are designed as fault containment units — independent power supplies, locations etc.
  - And ignoring high intensity radiated fields (HIRF) — and other initiators of correlated faults
- So we can duplicate the component and compare the outputs
  - Pass on the output when both agree
  - Signal failure on disagreement
- Under what assumptions does this work?

Example: Self-Checking Pair (ctd. 1)



- Controllers apply some control law to their input
- Controllers and distributor can fail
  - For simplicity, checker is assumed not to fail
  - Can be eliminated by having the controllers cross-compare
- Need some way to specify requirements and assumptions
- Aha! correctness requirement can be an idealized controller

Example: Self-Checking Pair (ctd. 2)

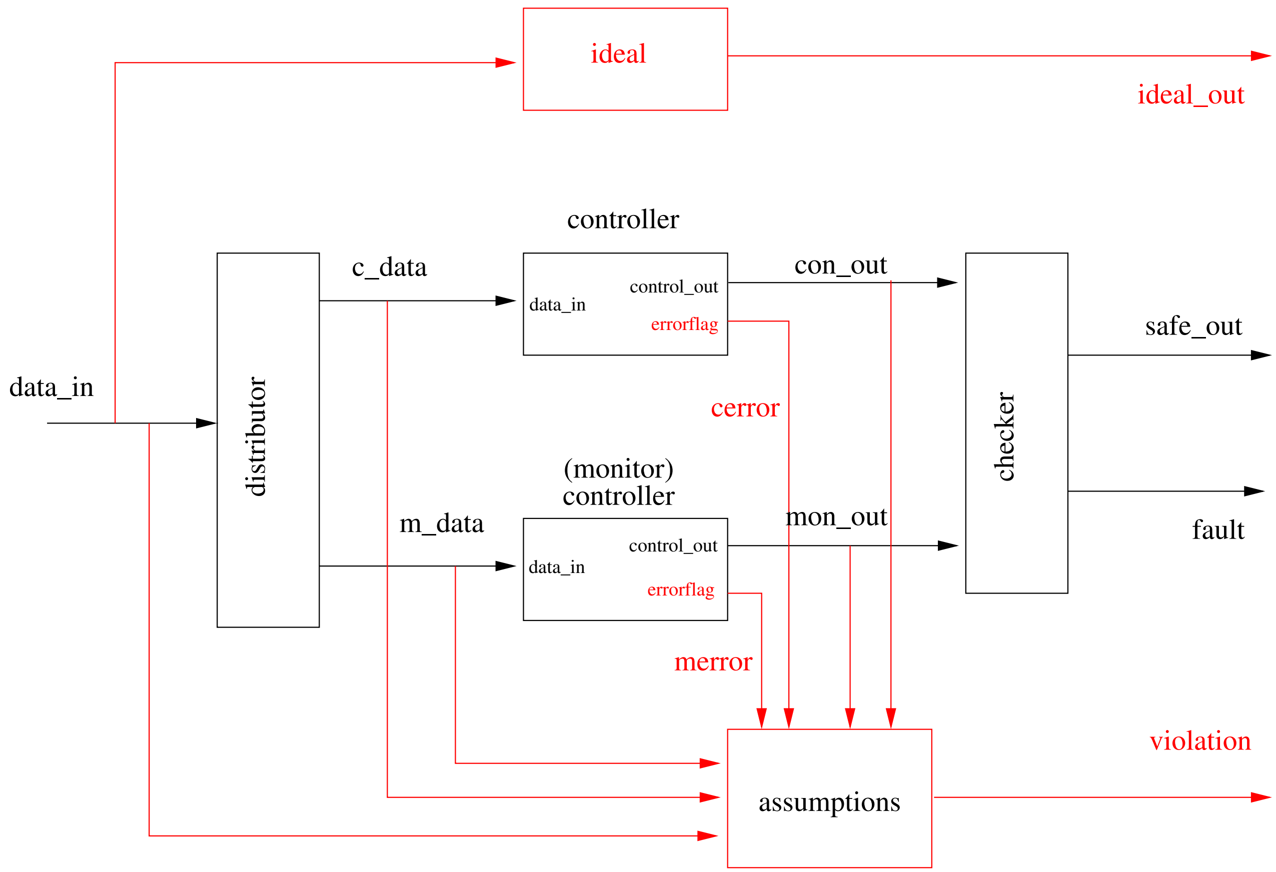


The controllers can fail, the ideal cannot

If no `fault` indicated `safe_out` and `ideal_out` should be the same

Model check for `G((NOT fault => safe_out = ideal_out))`

Example: Self-Checking Pair (ctd. 3)



We need assumptions about the types of fault that can be tolerated: encode these in assumptions synchronous observer

```
G(violation = down => (NOT fault => safe_out = ideal_out))
```

Synthesized Assumptions for Self-Checking Pair

- We will examine this example with the SAL model checker
- Initially, no assumptions
- Counterexamples help us understand what is wrong or missing
- Will discover four assumptions
- Then verify that the design is correct under these assumptions
- Then consider the probability of violating these assumptions and modify our design so that the most likely one is eliminated

selfcheck.sal: Types

```
selfcheck: CONTEXT =
BEGIN

sensor_data: TYPE;

actuator_data: TYPE;
init: actuator_data;

laws(x: sensor_data): actuator_data;

metasignal: TYPE = {up, down};
```

selfcheck.sal: Ideal Controller

```
ideal: MODULE =
BEGIN
INPUT
  data_in: sensor_data
OUTPUT
  ideal_out: actuator_data
INITIALIZATION
  ideal_out = init;
TRANSITION
  ideal_out' = laws(data_in)
END;
```

selfcheck.sal: Ordinary Controller

```
controller: MODULE =
BEGIN
INPUT
  data_in: sensor_data
OUTPUT
  control_out: actuator_data,    errorflag: metasignal
INITIALIZATION
  control_out = init;    errorflag = down;
TRANSITION
[ normal: TRUE -->
   control_out' = laws(data_in); errorflag' = down;
[] hardware_fault: TRUE -->
   control_out' IN {x: actuator_data | x /= laws(data_in)};
   errorflag' = up;
] END;
```

selfcheck.sal: Distributor

```
distributor: MODULE =
BEGIN
INPUT
  data_in: sensor_data
OUTPUT
  c_data, m_data: sensor_data
INITIALIZATION
  c_data = data_in; m_data = data_in;
TRANSITION
[  distributor_ok:  TRUE -->
    c_data' = data_in'; m_data' = data_in';
[] distributor_bad: TRUE -->
    c_data' IN {x: sensor_data | TRUE};
    m_data' IN {y: sensor_data | TRUE};
] END;
```

selfcheck.sal: Checker

```
checker: MODULE =
BEGIN
INPUT
  con_out: actuator_data,    mon_out: actuator_data
OUTPUT
  safe_out: actuator_data,   fault: boolean
INITIALIZATION
  safe_out = init;   fault = FALSE;
TRANSITION
safe_out' = con_out';
[
disagree:  con_out' /= mon_out' --> fault' = TRUE
[] ELSE -->
]
END;
```

selfcheck.sal: Wiring up the Self-Checking Pair

```
scpair: MODULE = distributor
 || (RENAME
      control_out TO con_out,
      data_in TO c_data,
      errorflag TO cerror
    IN controller)

 || (RENAME
      control_out TO mon_out,
      data_in to m_data,
      errorflag TO merror
    IN controller);

 || checker
```

selfcheck.sal: Assumptions

```
assumptions: MODULE =
BEGIN
OUTPUT
  violation: metasignal
INPUT
  data_in, c_data, m_data: sensor_data,
  cerror, merror: metasignal,
  con_out, mon_out: actuator_data
INITIALIZATION
 violation = down
TRANSITION
[ assumption_violation:
   FALSE % OR your assumption here (actually hazard)
      --> violation' = up;
[] ELSE --> ] END;
```

selfcheck.sal: Testing the Assumptions

```
scpair_ok: LEMMA
 scpair || assumptions || ideal |-
   G(violation = down
      => (NOT fault => safe_out = ideal_out));

% sal-inf-bmc selfcheck scpair_ok -v 3 -it
```

Assumption Synthesis: First Counterexample

- Both controllers have hardware faults
- And generate same, wrong result
- Derived hazard (assumption is its negation)

  `cerror' = up AND merror' = up AND con_out' = mon_out'`

  Assumption module reads data of different "ticks"; important to reference correct values (new state here)
- This hazard requires a double failure
  - Any double failure may be considered improbable
- Here, require double failure that gives same result
  - Highly improbable

Assumption Synthesis: Second Counterexample

- Distributor has a fault: sends wrong value to one controller
- The controller that got the good value has a fault, generates same result as correct one that got the bad input
- Derived hazard (assumption is its negation)

```
m_data /= c_data
  AND (merror' = up OR cerror' = up)
   AND mon_out' = con_out'
```

- Double fault, so highly improbable

Assumption Synthesis: Third Counterexample

- Distributor has a fault: sends (different) wrong value(s) to one or both controllers: Byzantine/SOS fault
- It just happens the different inputs produce same outputs
- Very dubious you could find this with a concrete model
  - Such as is needed for conventional model checking
  - Likely to use `laws(x) = x+1` or similar
- Derived hazard (assumption is its negation)

```
m_data /= c_data
  AND (merror' = down AND cerror' = down)
   AND mon_out' = con_out'
```

Assumption Synthesis: Third Counterexample (ctd.)

- The distributor could be as simple as a solder joint, how can it produce these failures?
- By adding resistance: one component may see weak voltage as 1, another as 0
- So actually quite plausible
- But fixable: pass inputs to checker
- Since the controllers are nonfaulty they correctly pass their different inputs to the checker
- This also reduces likelihood of the previous hazard, but does not eliminate it: the faulty controller could lie about its input

Assumption Synthesis: Fourth Counterexample

- Distributor has a fault: sends same wrong value to both controllers

- Derived hazard (assumption is its negation)

  ```
  m_data = c_data AND m_data /= data_in
  ```

- This one we need to worry about

- Byzantine/SOS fault at the distributor is most likely to generate the previous two cases
  - This is an unlikely random fault, but suggests a possible systematic fault

Assumption Synthesis Example: Summary

- We found four assumptions for the self-checking pair
  - When both members of pair are faulty, their outputs differ
  - When the members of the pair receive different inputs, their outputs should differ
    - When neither is faulty: can be eliminated
    - When one or more is faulty
  - When both members of the pair receive the same input, it is the correct input
- Can prove by 1-induction that these are sufficient
  - `sal-inf-bmc selfcheck scpair_ok -v 3 -i -d 1`
- One assumption can be eliminated by redesign, two require double faults
- Attention is directed to the most significant case

The Big Picture

- Formal verification does not cover everything needed to certify safety
- For example, there are probabilistic elements concerning random failures and vulnerabilities in the verification itself (correctness of the requirements, completeness of hazard discovery)
- These are addressed in the Safety Case, generally organized around Claims, Argument, Evidence
- Sometimes Toulmin's style of argument is advocated here
  - "argumentation" is a field of its own, distinct from logic
- Certainly need some kind of probabilistic logic
  - Carnap, BBNs, Dempster-Shafer etc.
- Many opportunities for research here

Conclusion

- The problem of composing truly safe systems from components throws many of the issues concerning component and system design and verification into sharp relief
- I hope I have illustrated some of these
- And excited you about the opportunities for research here