Invited talk for 15th Brazilian Symposium on Formal Methods (SBMF), Natal, 25 Sep 2012

The Versatile Synchronous Observer

John Rushby

Computer Science Laboratory
SRI International
Menlo Park CA USA

Model Checking

- Informally, model checking means fully automated FM
- But it's called model checking because we check
  - Whether our system (or program or design), represented as a state machine
  - Is a Kripke model of
  - Our specification, represented as a temporal logic formula
- Typically, the specification is translated into a state machine
- And composed with the system state machine
- And we try to prove that all reachable states satisfy the specification, or we exhibit a counterexample
- Automated by explicit state (exhaustive simulation, e.g., SPIN), symbolic finite state methods (BDDs, or BMC and k-induction with SAT, e.g., NuSMV), or symbolic infinite state methods (BMC and k-induction with SMT, e.g., SAL)

Safety and Liveness

- If the specification is a liveness/eventuality property (typically, one involving the **F** or $\Diamond$ modalities)
- Then it will be translated to a Büchi automaton, and the checker will apply special acceptance rules
  - Must reach a goal state infinitely often
- But for safety properties, it is just a regular automaton, i.e., state machine
- In practice, we only care about safety properties
  - Note that bounded liveness is a safety property

Synchronous Observers

- For safety properties, instead of writing the specification as a temporal logic formula and translating it to a state machine
- We could just write the specification directly as a state machine
- Specifically, a state machine that is synchronously composed with the system state machine
- And that observes its state variables
- And either signals an alarm if the intended behavior is violated, or OK as long as it is not
- This is called a synchronous observer
- Then we check that alarm or not OK are unreachable
  - `check: FORMULA (system || observer) |- G(NOT alarm)`
  - `check_alt: FORMULA (system || observer) |- G(OK)`

Origins

- Both the concept and the term synchronous observer were introduced in the context of the synchronous languages developed in France
- In particular, by the Lesar model checker for the language Lustre

Benefits

- We only have to learn one language
    - The state machine language
- Instead of two
    - State machine plus temporal logic specification language
- And only one way of thinking
- Well, not quite
    - System generates behavior and observer recognizes it

The Versatility of Synchronous Observers

- There are several other uses for synchronous observers
- I'll describe four, there are probably more
  1. Increased expressivity
  2. Specifying/discovering assumptions
  3. And axioms
  4. Test generation

Increased Expressivity via Synchronous Observers (1)

- Typical state machine language allows new values of variable to be defined in terms of the old (notation here is SAL)
  - e.g., `x' = x + y`
    or `x' IN {a: nat | a >= 25 AND a <= 50}`
- What if we want to specify that the new value of `x` is simply larger than the old?
- Some languages allow for this in nondeterministic assignments
  - e.g., `x' IN {a: nat | a > x}`
- And some by allowing new values to appear in guards
  - e.g., `(x' > x) --> x' IN {a: nat | TRUE}`
- But one method that always works is to specify it using a synchronous observer. . .

Increased Expressivity via Synchronous Observers (2)

- First, in main system, make an unconstrained assignment to `x`
  - `x' IN {a: nat | TRUE}`
- Then, in a synchronous observer for constraints, we enforce the desired relation (using `AOK` as our flag variable)
  - `NOT (x' > x) --> AOK' = FALSE`

  (if new variables are not allowed in the guards, then we will need to introduce history variables)
- Then we model check for whatever property `p` we had in mind, only in cases where `AOK` is `TRUE`
  - `check: FORMULA (system || constraints) |- G(AOK => p)`, or
  - `check: FORMULA (system || observer || constraints) |- G(AOK => OK)`
- This method is particularly useful when need to update multiple variables in a way that enforces a relation on them
  - cf. relational abstraction for hybrid automata

Fragment of Constraints from FMIS 2011 Example

```
INITIALIZATION
   ok = TRUE;
TRANSITION
[  actual_mode = op_des AND pitch > 0 --> ok' = FALSE;
[] actual_mode = op_clb AND pitch < 0 --> ok' = FALSE;
[] actual_mode = vs_fpa AND fcu_fpa <= 0 AND pitch > 0
    --> ok' = FALSE;
[] actual_mode = vs_fpa AND fcu_fpa >= 0 AND pitch < 0
    --> ok' = FALSE;
[] pitch > 0 AND altitude' < altitude --> ok' = FALSE;
[] pitch < 0 AND altitude' > altitude --> ok' = FALSE;
[] pitch=0 AND altitude' /= altitude --> ok' = FALSE;
[] ELSE -->
] END;
```

Synchronous Observers for Assumptions

- Most properties are not expected to be true unconditionally
- They are expected to be true only in environments that satisfy certain assumptions
- Assumptions should generally be stated as constraints, not by specifying an ideal environment
  - Our job is to specify the environment, not implement it
- Synchronous observers can do this
  - `NOT assumption`$_i$ `--> AOK' = FALSE`
- Illustrated in the previous example

Synchronous Observers for Axioms (1)

- The biggest advance in formal methods in the last 20 years has been the development of high-performance SMT solvers
  - Solvers for Satisfiability Modulo Theories
- Roughly, these combine decision procedures for useful theories like equality with uninterpreted functions, linear arithmetic on integers and reals, arrays, and several others
  - These work on conjunctions of formulas
- With SAT solvers
  - These handle propositionally complex formulas
- The combination uses an abstraction/refinement/learning loop, plus a lot of engineering
- SMT brings effective automation to many formal methods
- TBD: nonlinear arithmetic, quantifiers, and lemma (invariant) generation

Synchronous Observers for Axioms (2)

- One of the disadvantages of model checking compared to theorem proving in a system like PVS is that model checking requires us to be too explicit
  - For most model checking technologies, the system has to be a (possibly nondeterministic) implementation
- Suppose we want to examine the bypass logic of a CPU pipeline; typically want to prove the sequence of values out of the pipelined implementation is same as nonpipelined one
- There's an ALU at end of the pipeline; we don't care what fn's it computes, just that at step i it does some f_i`(a, b)`
- But to model check, must put a specific circuit there
  - e.g., an adder: and some bugs may then go undetected because of the special properties of that implementation (e.g., commutativity, associativity)

Synchronous Observers for Axioms (3)

- The reason theorem provers are more attractive than model checkers for these kinds of situation is that they allow use of uninterpreted functions: $f(x)$ where we know nothing about f
- Can constrain f by adding axioms
  - e.g., `x > y => f(x) > f(y)`
- SMT solvers decide this theory
- And SMT solvers can be used for model checking via BMC and k-induction
- So now we can model check over specifications that use uninterpreted functions etc.
  - Here, model checking is used to mean fully automatic
- This technology is called infinite bounded model checking or infBMC ('cos some of the theories are over infinite models)

Synchronous Observers for Axioms (4)

- But how do we convey the axioms about our uninterpreted functions to the SMT solver underlying our infBMC?
- Synchronous observers!
- As before, just check for violations of the axioms
  - `NOT axiom`$_i$ `--> AOK' = FALSE`
- Whew!
- That was a lot of setup to get to a simple conclusion
- Let's extract more from the same setup

Discovering Assumptions with Synchronous Observers

- In civil aircraft, all accidents and incidents caused by software are due to flaws in the system requirements specification or to gaps between this and the software specification
  - i.e., none are due to coding errors
- Modern system requirements specifications look a lot like software: lots of case analysis
- But are very abstract (box and arrow diagrams)
- There's no accepted technology for analyzing these
- But infBMC can do it
- Use uninterpreted functions for the boxes and arrows
- Incrementally add constraints/axioms to a synchronous observer
- Until the desired properties are satisfied

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, but we still have to worry about random faults
  - e.g., when an α-particle flips a bit in instruction counter
- Our goal here is to design a component that fails cleanly in the presence of random faults

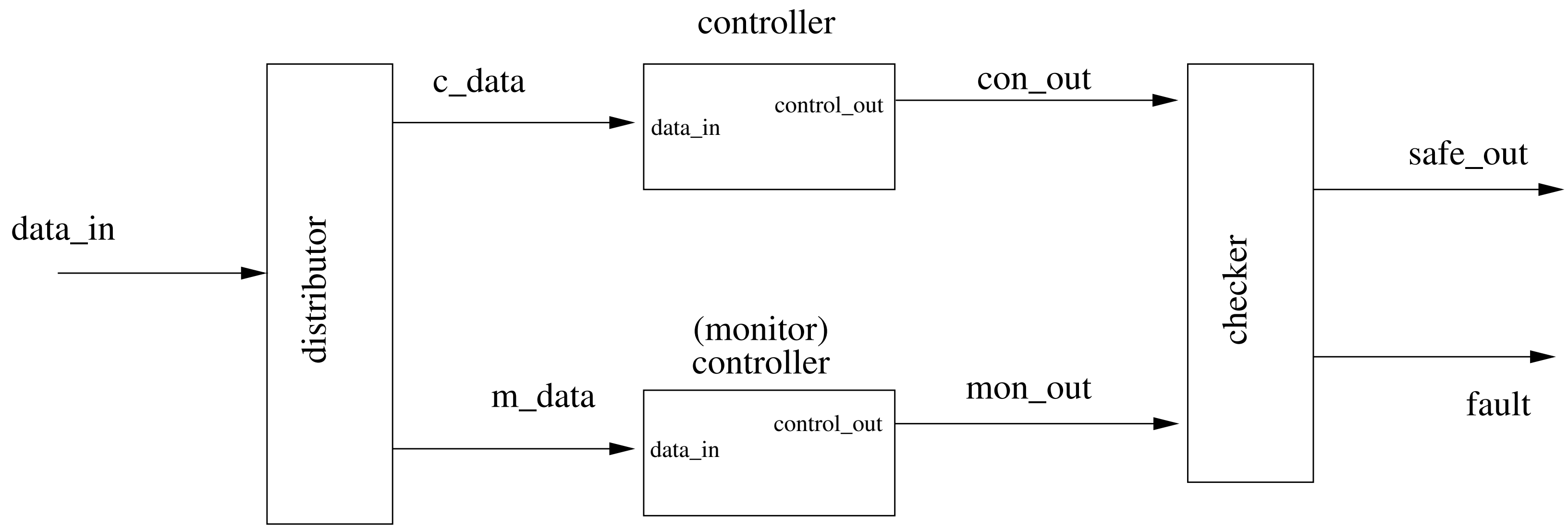
Example: Self-Checking Pair (1)

- 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

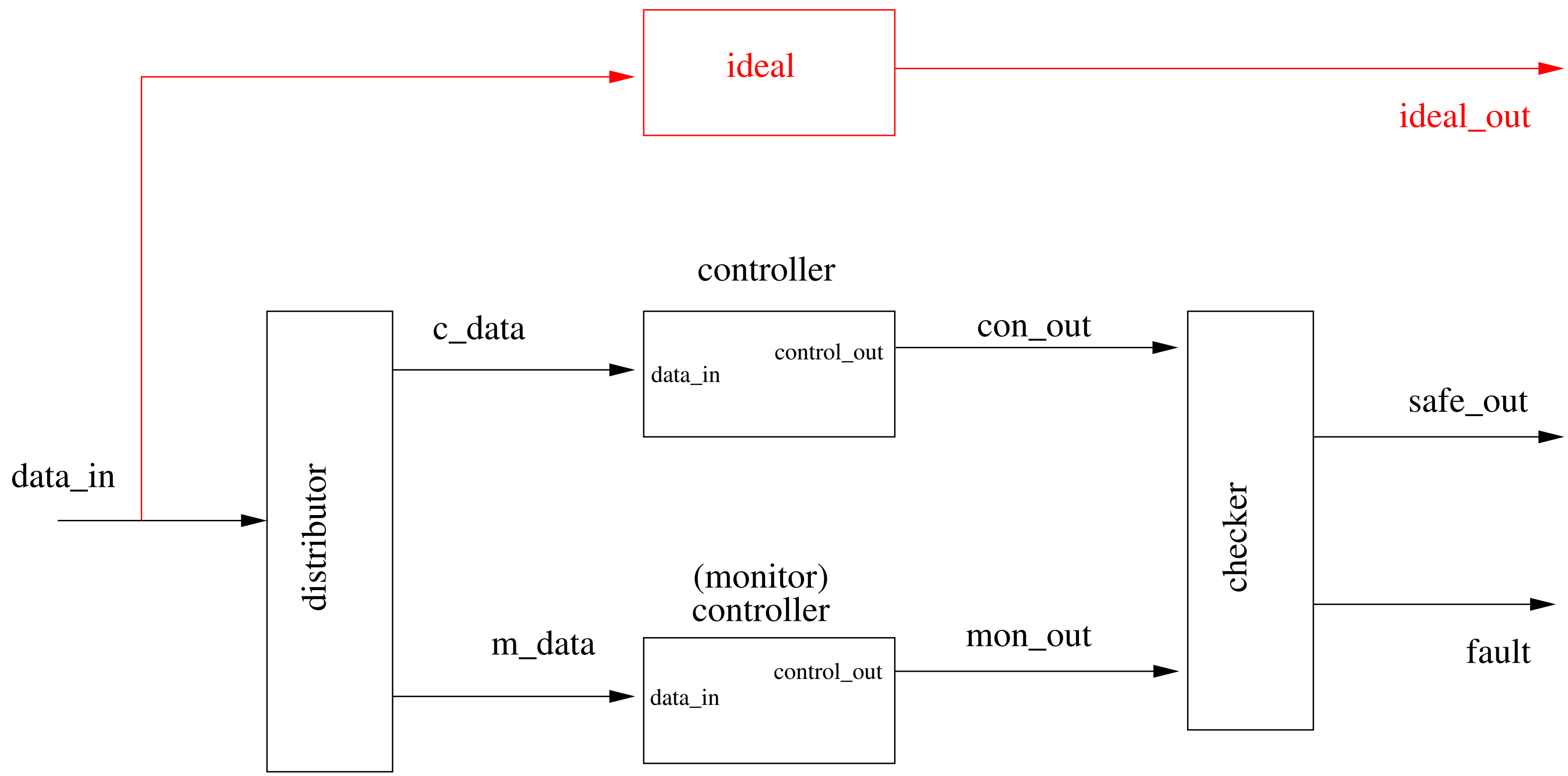
- Under what assumptions does this work?

Example: Self-Checking Pair (2)



- 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 (3)

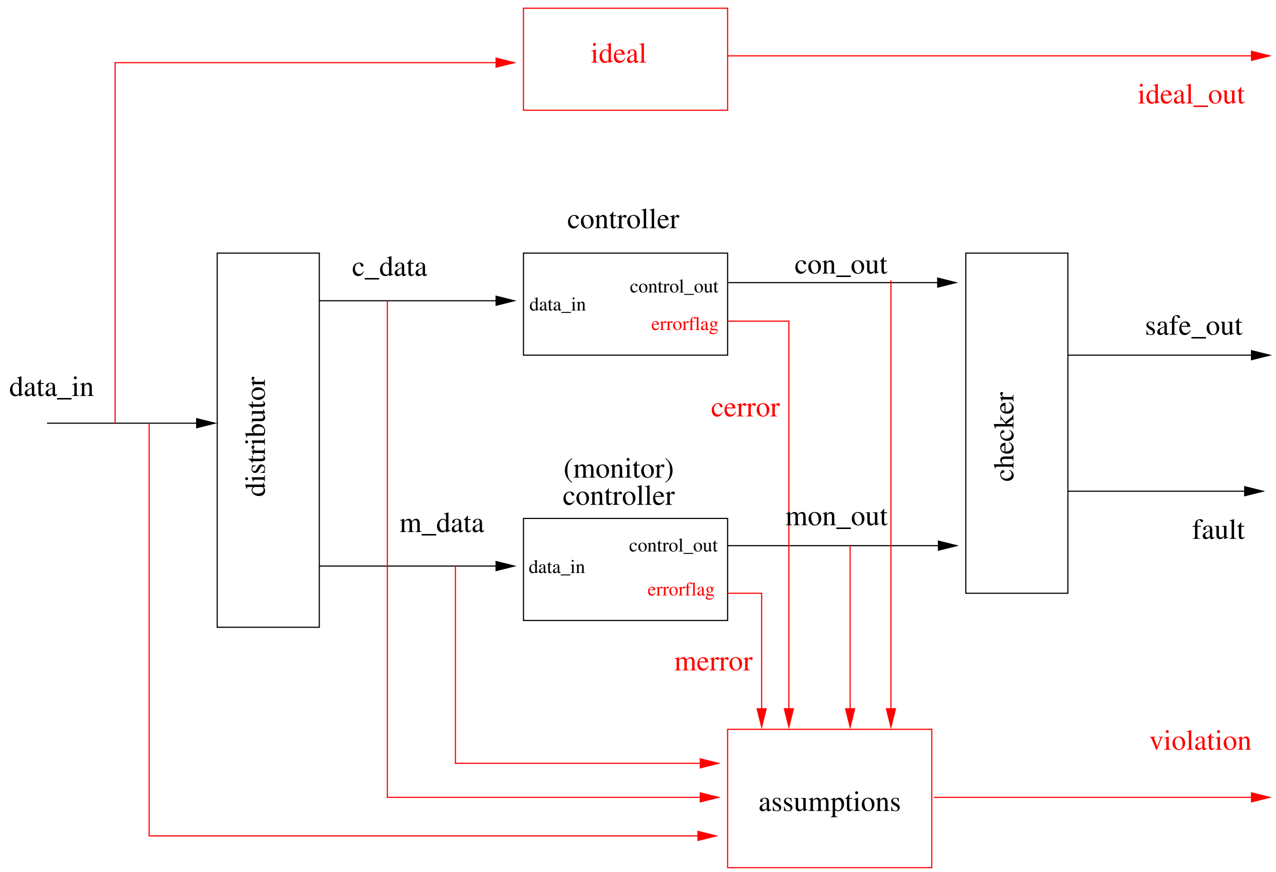


The controllers can fail, the ideal cannot

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 (4)



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

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

Example: Self-Checking Pair (5)

- Find 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
- One assumption can be eliminated by redesign
- Two require double faults
- Attention is directed to the most significant case

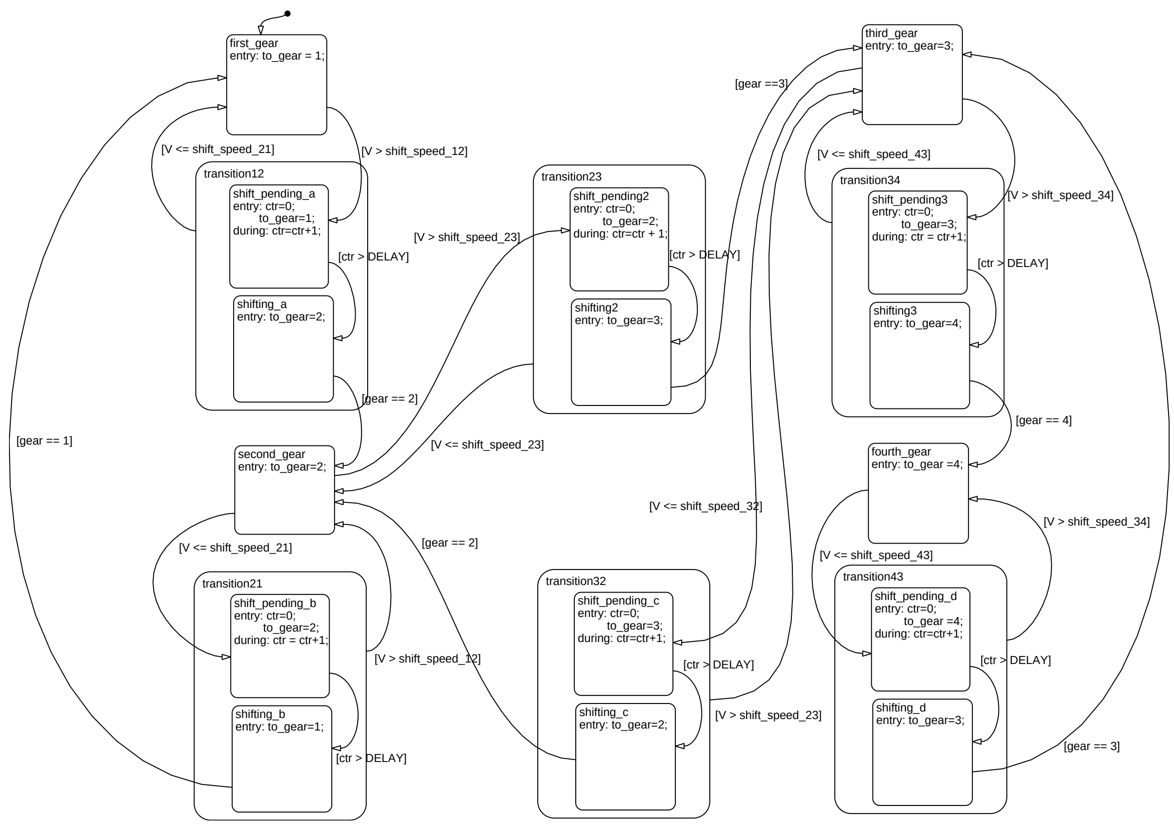
Test Generation

- Model checkers can be used for test generation
- e.g., to generate a test that reaches a target state characterized by property `p` just check for `NOT p`
  - `test: FORMULA system |- G(NOT p)`

  The counterexample generated by the model checker is a test scenario to reach the target state
- Can modify a model checker to generate single (long) counterexample to reach multiple targets
  - SAL-ATG does this
- But a cool alternative is to write a synchronous observer "tester" module that raises a flag `TOK` when it has observed a scenario that satisfies the test purpose
- Then model check for `NOT TOK`
  - `test: FORMULA (system || tester) |- G(NOT TOK)`

Example: Shift Scheduler (1)

This is the Simulink/Stateflow design for the shift scheduler of an automatic transmission used in Ford cars



We want a test scenario that takes it through all its states

Example: Shift Scheduler (2)

- One input is the `gear` currently selected by the gearbox
- Tests often change this discontinuously (e.g., 1, 3, 4, 2)
- Can easily establish the test purpose to change only in single steps, and to change at every step
- Create a tester module whose body is

```
OUTPUT
 moving, continuous: BOOLEAN
INITIALIZATION
 moving = TRUE; continuous = (gear=1);
TRANSITION
 moving' = moving AND (gear /= gear');
 continuous' = continuous
  AND (gear - gear' <= 1)AND (gear' - gear <= 1);
```

- Then model check for the negation of these
  - `test: FORMULA (system || tester) |- G(NOT (moving AND continuous))`
- Actually, also need to check for a `DONE` flag on state coverage

Observations

- Instead of constructing behavior as in a system specification
- A synchronous observer recognizes it
- And the model checker synthesizes the behavior for us
- May be costly with an explicit state model checker
  - Has to generate many behaviors, then throw them away

  But OK for symbolic ones
- Recall the examples
  1. Increased expressivity: useful
  2. Specifying/discovering assumptions: awesome!
  3. And axioms: creates new opportunities
  4. Test generation: indispensable
- But wait, there's more!

Synchronous Observers at Runtime

- Instead of just using the synchronous observer in analysis
- We could use it at runtime: as a monitor
  - This is often called runtime verification
- It's particularly interesting in safety critical applications, where you need extreme reliability
  - One operational "channel" does the business
  - Simpler monitor channel can shut it down if `not OK`
- Used in airplanes (ARP 4754)
- Turns malfunction and unintended function into loss of function
  - Which is dealt with OK by higher-level fault handling

  Also prevents transitions into bad states

Reliability of Monitored Systems (1)

- The most critical aircraft software needs failure rates below 10^{-9} per hour sustained for 15 hours
- Suppose the failure rate of the operational system is 10^{-4} and that of the monitor is 10^{-5}, does that give us 10^{-9}?
- No! Failures may not be independent
  - Failure of one channel probably indicates a hard demand
- No good way forward
  - Need "covariance of the difficulty function"

Reliability of Monitored Systems (2)

- But the monitor could be simple enough that it is formally verified or synthesized
- Claim is not that it is reliable but that it is perfect. . . probably
  - Perfection means will never have a failure in operation
  - Failure is defined wrt. system requirements, not software requirements, hence differs from correctness
- Attach subjective probability to likelihood of perfection
- Theorem: probability of perfection of the monitor is conditionally independent of the failure rate of the primary
- So if the monitor has probability of imperfection of 10^{-5}, we do get 10^{-9} overall!

Reliability of Monitored Systems (3)

- Lots of technical details omitted here
- This analysis is aleatoric, need the epistemic assessment
- And is 10^{-5} credible as a probability of imperfection?
- Monitor may go off when it should not (Type 2 failure)
- But the basic idea is sound (IEEE TSE November 2012)
- Idea is that you monitor the system specification
  - Get this right by assumption synthesis etc.
- Whereas the operational system is built to the software requirements specification
- Recall, all aircraft incidents due to problems precisely here
- So this approach precisely addresses most vulnerable point

Conclusions

- Synchronous observers are a fairly obvious idea
- But I don't think their versatility is widely appreciated
- So I hope to have given you some ideas for novel ways to exploit them, and invite you to think of more
- Can also be used at runtime, interesting reliability results via probability of perfection (relates assurance to reliability)
- More generally, the power of modern tools like SMT solvers and infBMC is such that it often makes sense to specify required behavior by means of a recognizer, or in terms of constraints, rather than by a constructive specification
  - Let the automation synthesize the behavior
- The next step is to let the automation synthesize the constructive specification or implementation from constraints
- For that, need to develop effective Exists-Forall SMT solvers