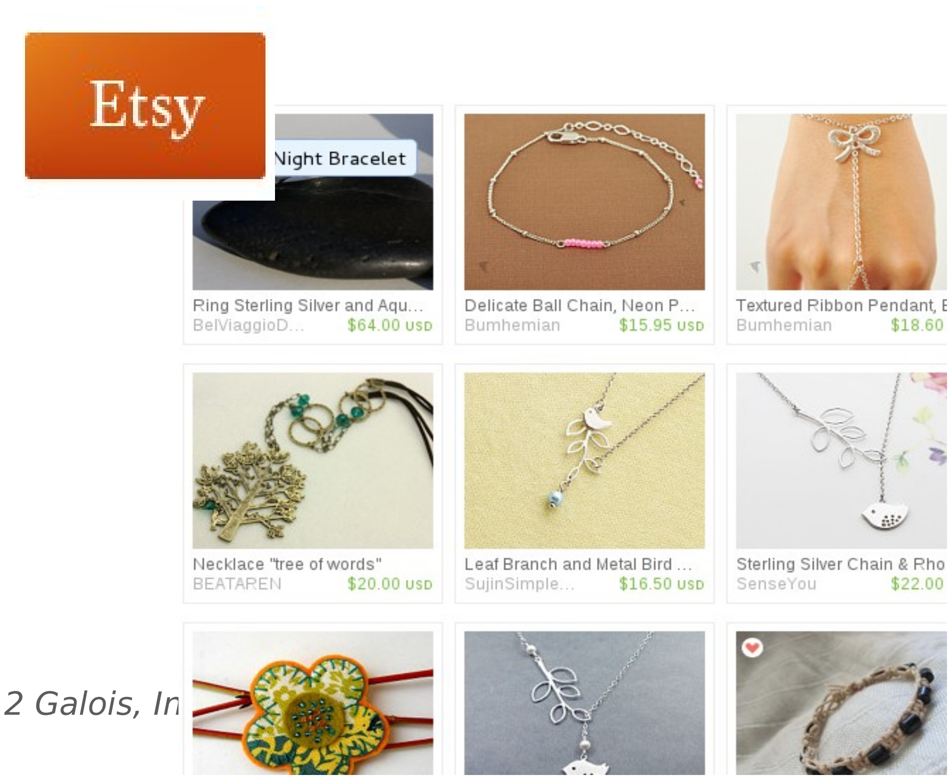
Experience Report: a Do-it-Yourself High-Assurance Compiler

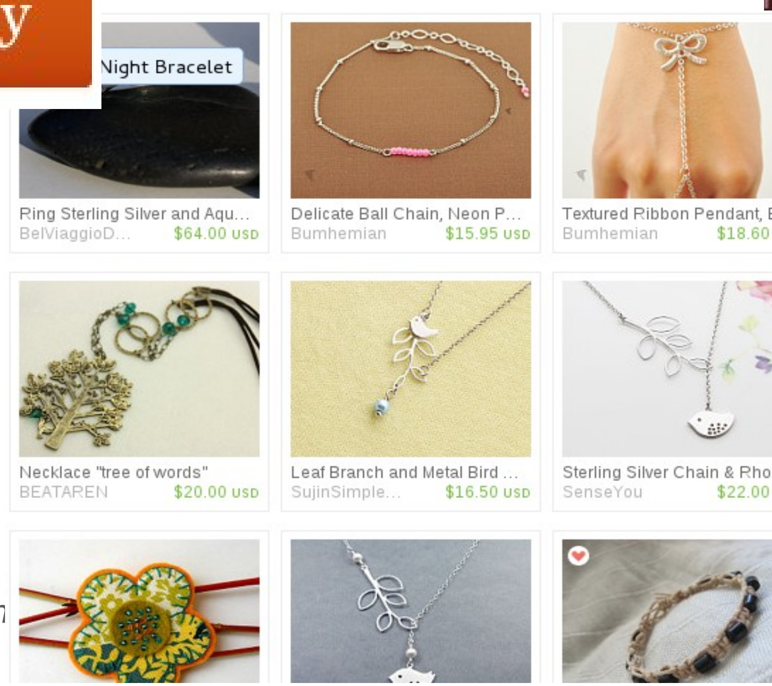
Lee Pike Galois, Inc. <leepike@galois.com>
Nis Wegmann University of Copenhagen
Sebastian Niller Unaffiliated
Alwyn Goodloe NASA Langley Research Center

Do-It-Yourself

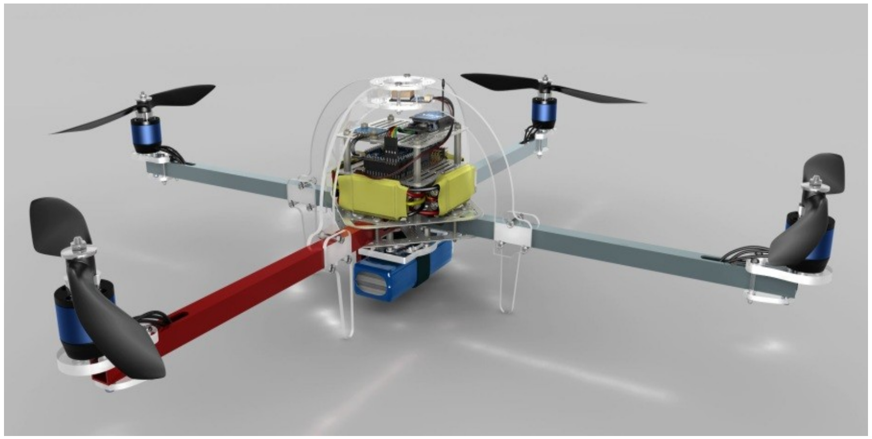


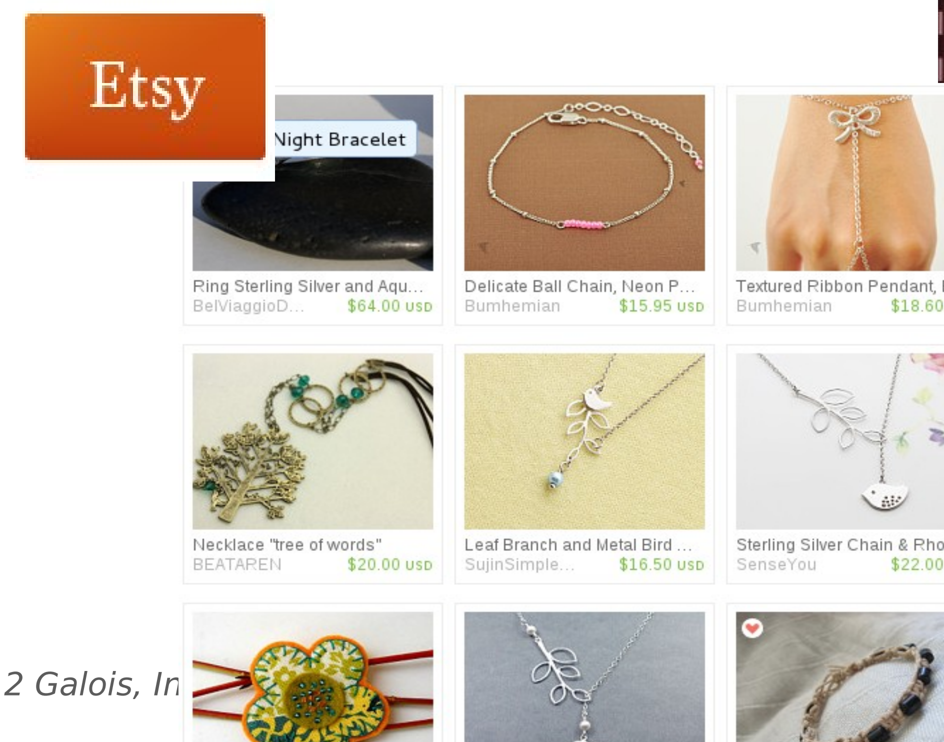
Do-It-Yourself

Etsy

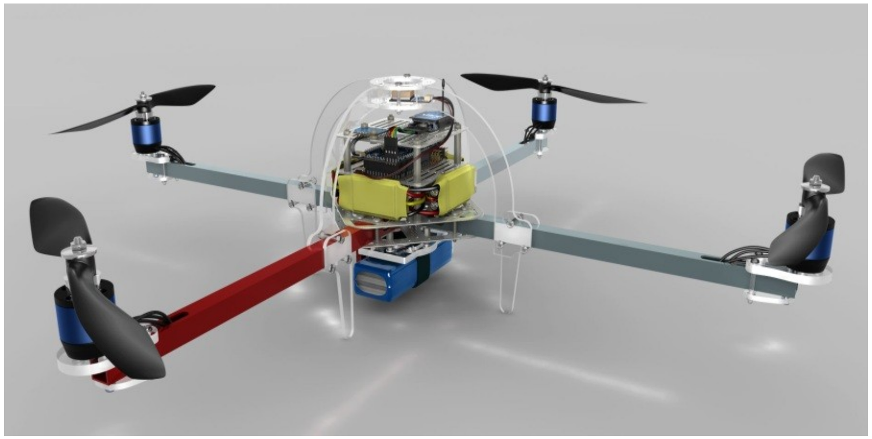


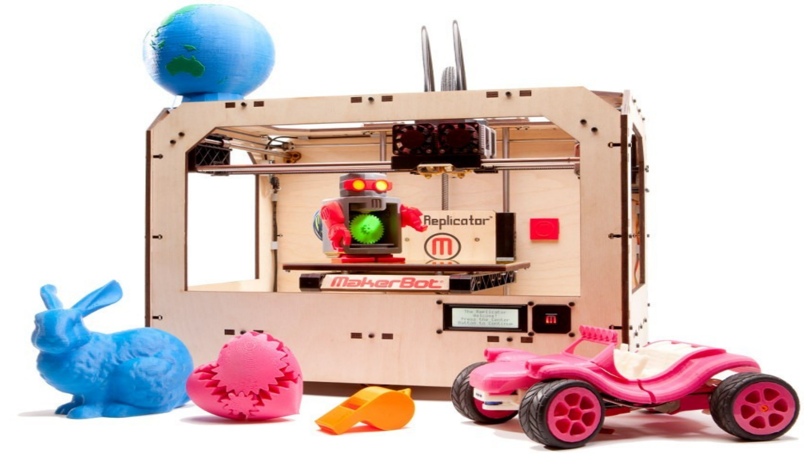
Do-It-Yourself

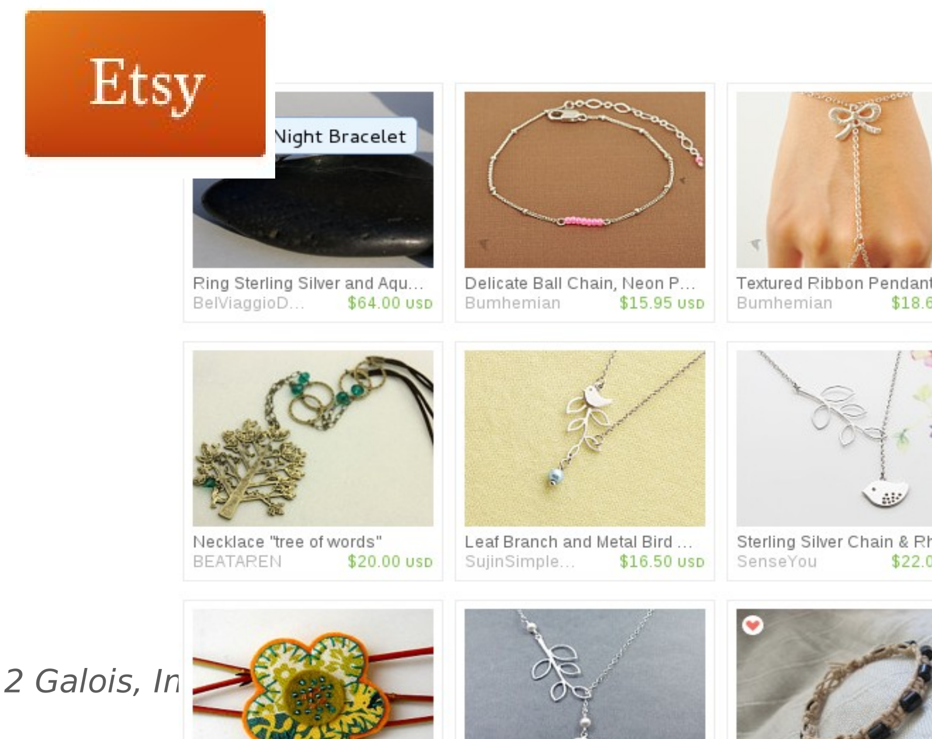




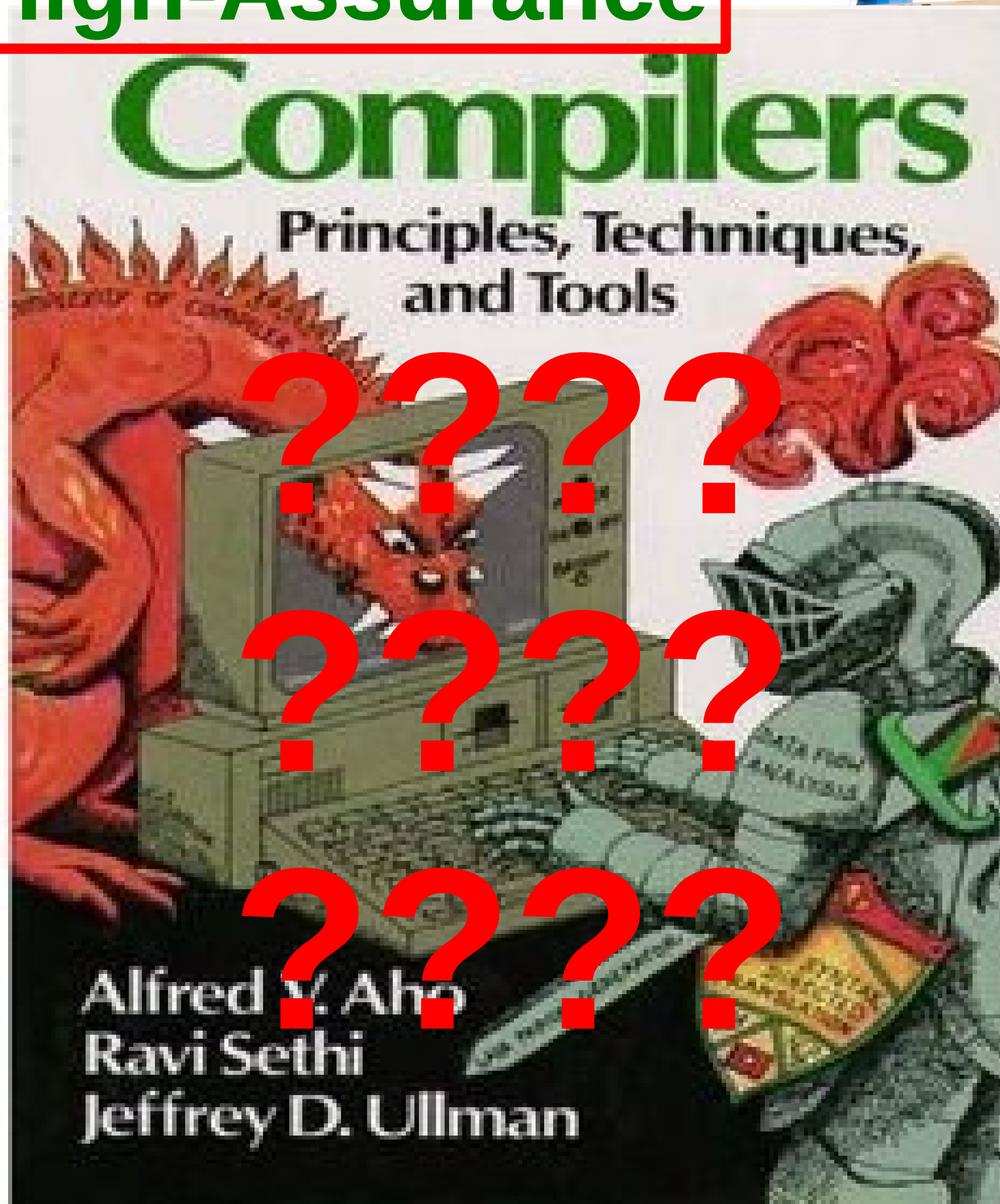
Do-It-Yourself







High-Assurance



????
????
????

3 Not-So-Secret Weapons

1. Embedded domain-specific languages (EDSLs)
2. A *verifying* (not verified) compiler approach
3. Open source testing/verification libraries & tools

National ??

and Space Administration

National Aeronautics and Space Administration

Copilot: a Run-Time Monitoring DSL

- Embedded DSL in Haskell
- Synthesize monitors for real-time embedded systems
- Stream language
- Generates Misra-like C
- Constant time, constant memory
  - Synthesized scheduler
  - No RTOS needed

Sample Copilot specification

Haskell

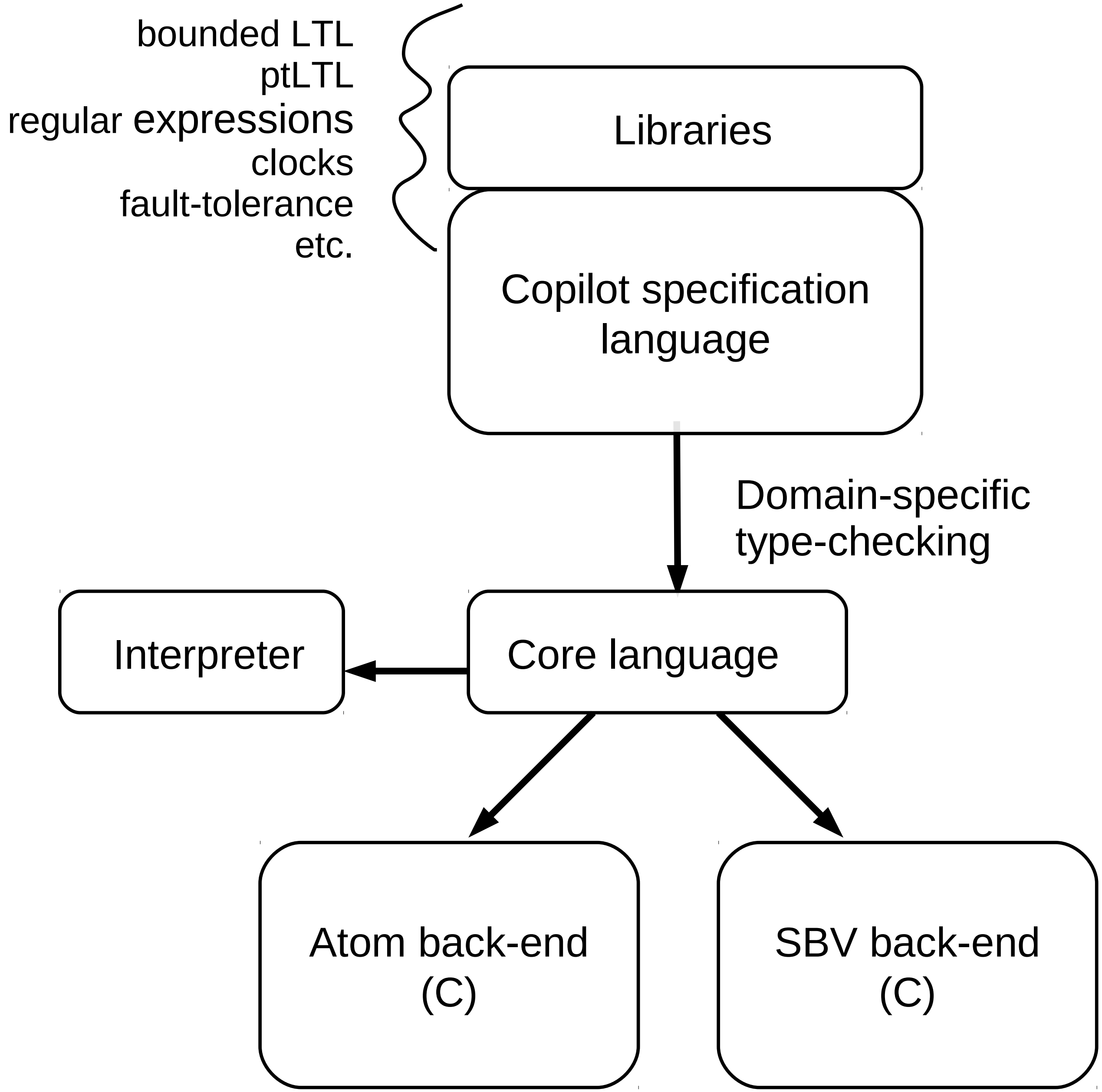
```
fib :: [Word32]
fib =  [0, 1] ++ zipWith (+) fib (drop 1 fib)
```

Copilot

```
fib :: Stream Word32
fib =  [0, 1] ++ (fib + drop 1 fib)
```

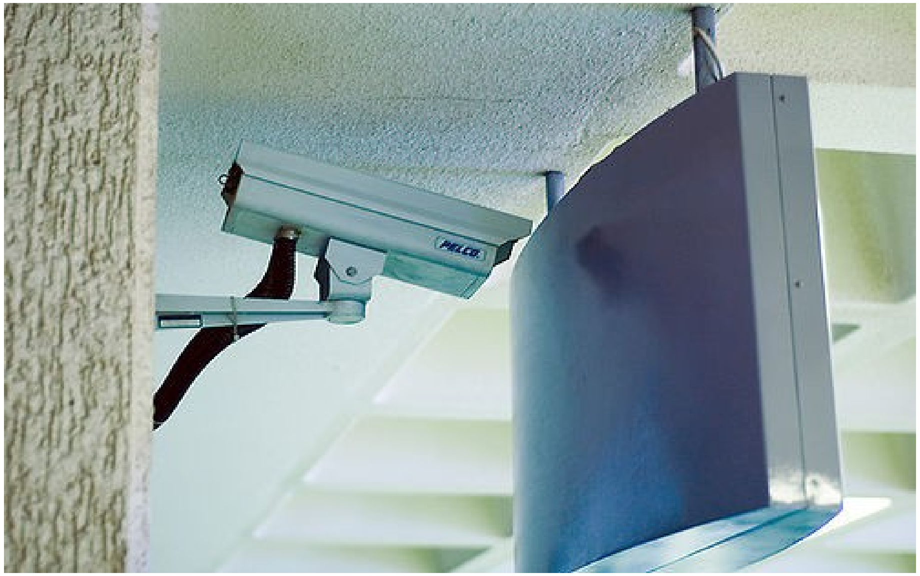
Special constructs for input (*sampling*) and output (*triggers*)

Copilot Architecture



Lessons in DIY Assurance

- Who monitors the monitor?



- Challenges:
  - EDSLs encourage rapid language design changes
  - Industrial work often doesn't “pay” for assurance (but wants it)

Lessons in DIY Assurance

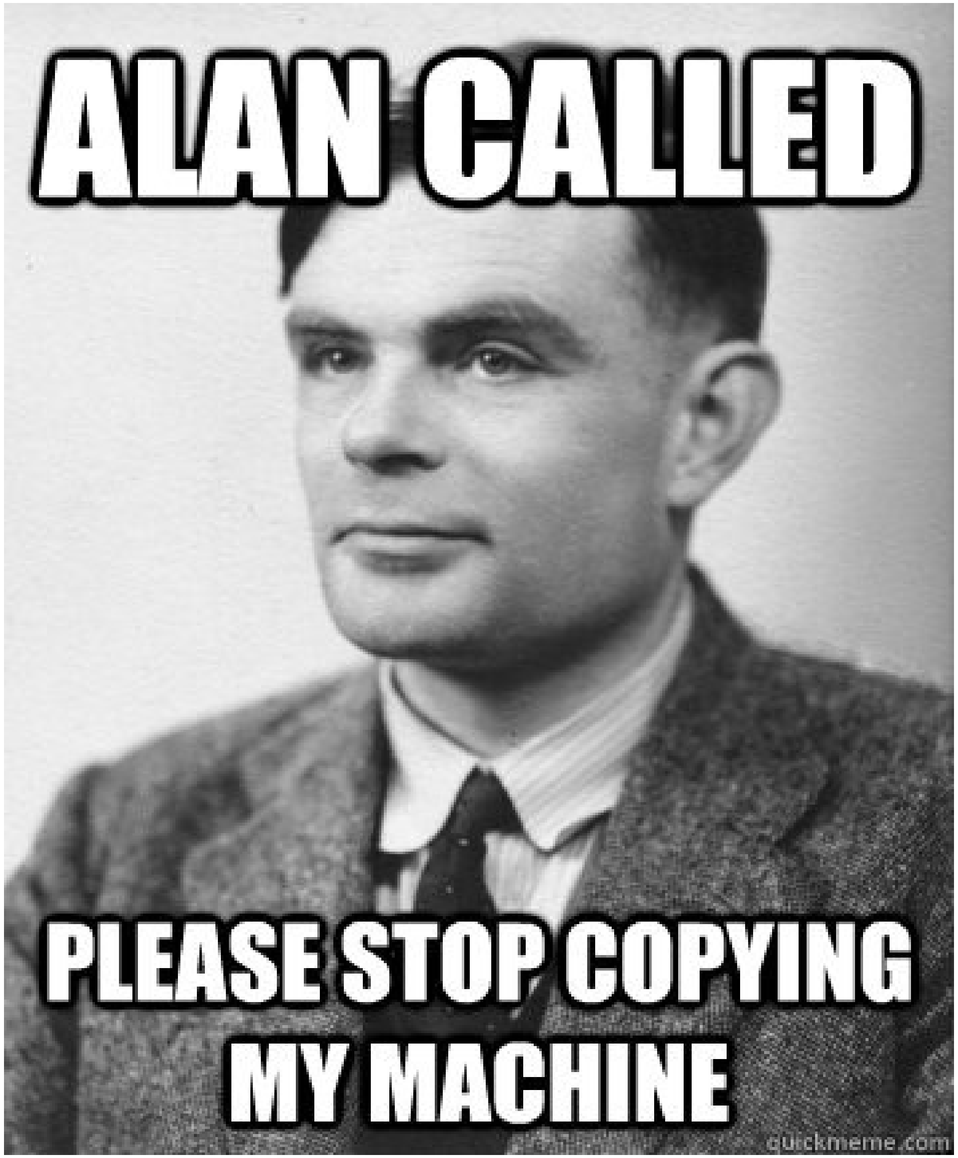
Solution: DIY assurance

- Turing *in*complete DSLs, Turing complete macros
- Multi-level type-checking
- Cheap testing & proofs
- Unified host language

Lesson #1: Turing-Incompleteness

Turing *in*completeness means:

- Compiler writing is simplified
- Compiler reasoning is better (e.g., termination analysis)
- Security is improved
- Automated verification has a chance of working!



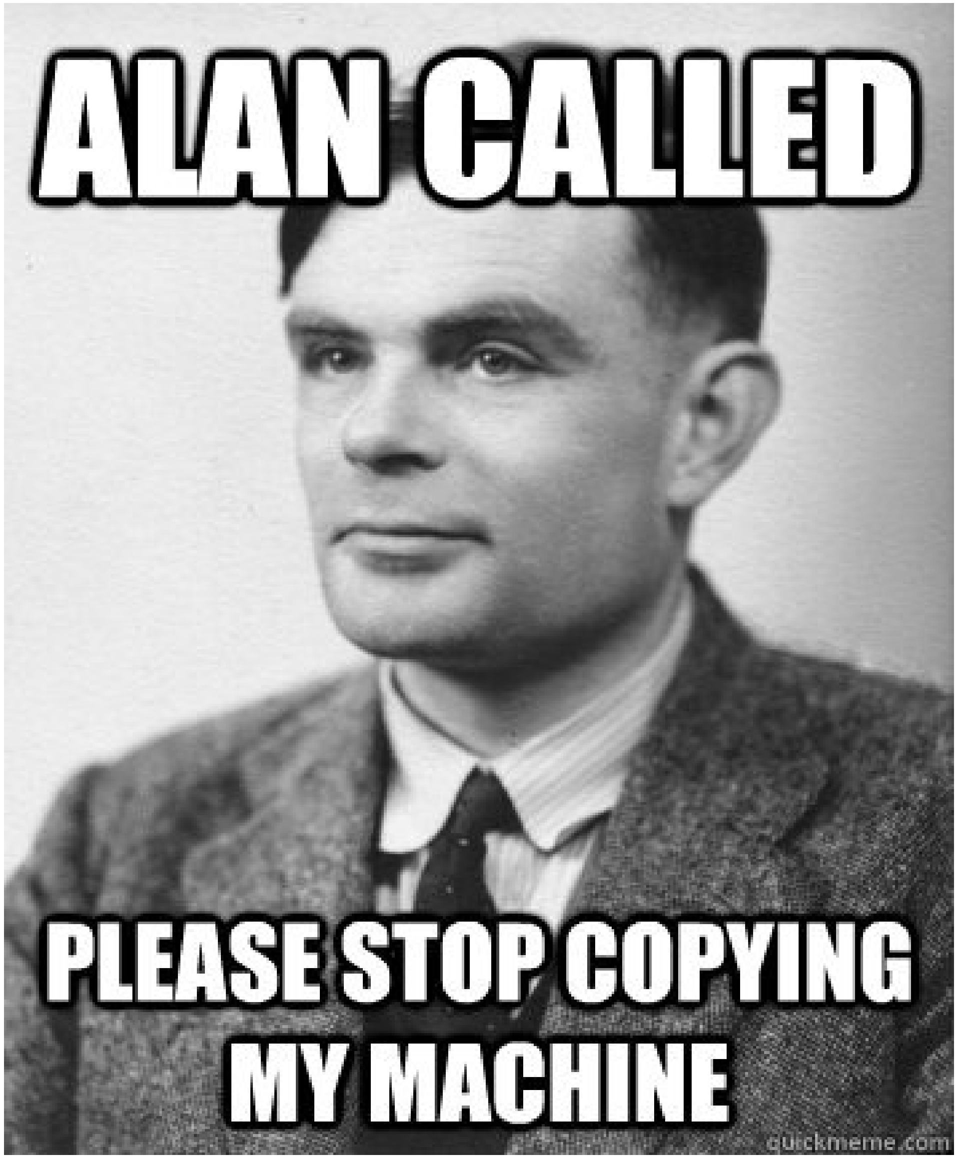
Lesson #1: Turing-Incompleteness

Turing *in*completeness means:

- Compiler writing is simplified
- Compiler reasoning is better (e.g., termination analysis)
- Security is improved
- Automated verification has a chance of working!

Have your cake and eat it, too:
In an embedded DSL, the *host* language is Turing-complete!

Programs specialized at compile time.



Lesson #2: Multi-Level Type-Checking

- Lean on Haskell's type system in the (DSL's) compiler's internal representations: e.g., GADTs
  - Leave the type system twice:
    - Pretty-print C
    - Translating between EDSLs (type-safe dynamic typing*).
  - And ensure you aren't abusing it: **Safe Haskell**

*Baars and Swierstra. *Typing dynamic typing*. ICFP 2002.

Lesson #2: Multi-Level Type-Checking

- Lean on Haskell's type system in the (DSL's) compiler's internal representations: e.g., GADTs
  - Leave the type system twice:
    - Pretty-print C
    - Translate between EDSLs (type-safe dynamic typing*).
  - And ensure you aren't abusing it: **Safe Haskell**

- Then a little domain-specific type-checking:
  - Productiveness:

    Rejected:

    ```
    x :: Stream Word64
    x = [0] ++ drop 1 x
    ```

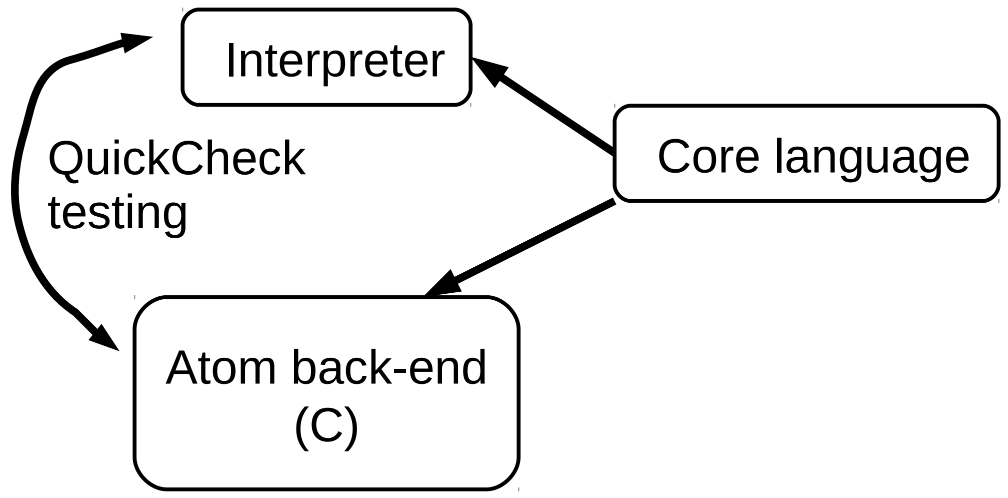
  - Inputs are consistently typed (e.g., external functional calls)

*Baars and Swierstra. *Typing dynamic typing*. ICFP 2002.

Lesson #3: Cheap Testing & Proofs

QuickCheck:

- Small DSLs make program generation easy with good coverage
- Test ~1.5M programs/day



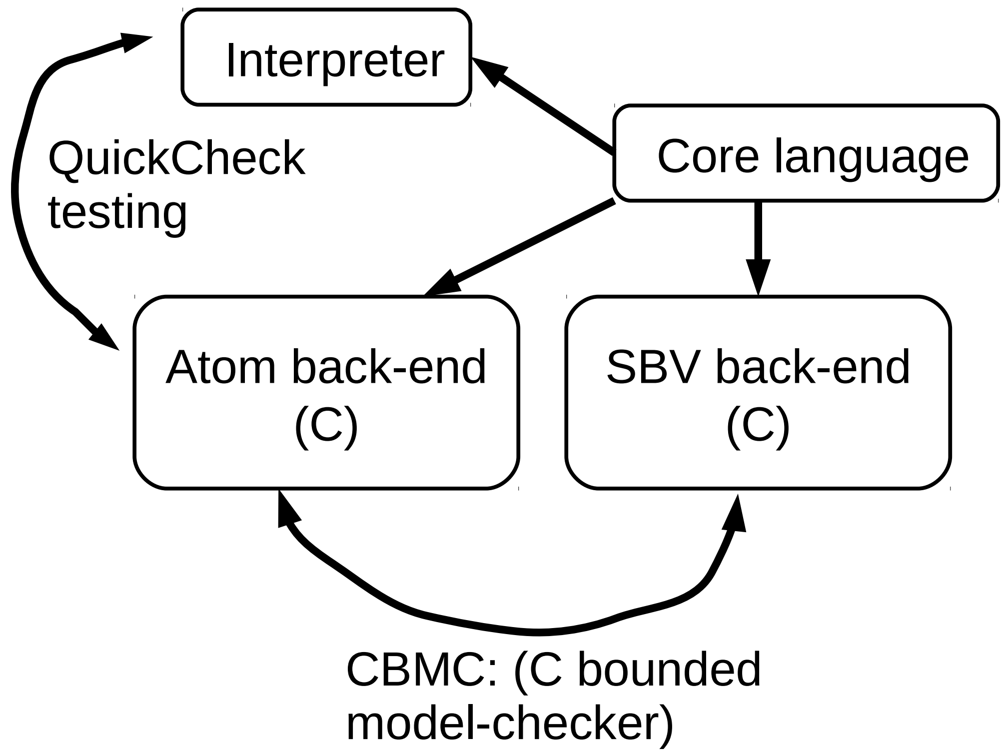
Lesson #3: Cheap Testing & Proofs

QuickCheck:

- Small DSLs make program generation easy with good coverage
- Test ~1.5M programs/day

Then **prove** back-ends agree:

- Model-checking works (better) with Turing incomplete DSLs
- EDSL simplifies driver generation



http://www.cprover.org/cbmc/

Lesson #4: a Unified Host Language

Embedded DSLs are a paradigm shift for safety-critical languages

- Fewer front-end, type-checker bugs
- “Bolting-on” new tools *within* the type system (no marshalling)
- The macro language is a build system, too!

Conclusions

Conclusions

Verified compiler

- Expensive
- Specialized skills
- Hard to make repairs
- But flawless when it works

Conclusions

Verified compiler

- Expensive
- Specialized skills
- Hard to make repairs
- But flawless when it works

src: http://designthatmatters.org/news/press/dtm-in-the-news/

Conclusions

Verified compiler

- Expensive
- Specialized skills
- Hard to make repairs
- But flawless when it works

src: http://designthatmatters.org/news/press/dtm-in-the-news/

DIY assurance

- Cheap
- Quick to build
- Easy to repair
- An “90% solution”

Monitoring constraints

Goal: run-time monitors for software-intensive embedded systems

Runtime monitoring for real-time embedded systems should satisfy the **FaCTS**:

- **F**alse-positives: don’t change the target’s behavior
- **C**ertifiability: make software re-certification easy

  Don't go changing sources
- **T**iming: don’t interfere with the target’s timing
- **S**WaP: don’t exhaust size, weight, power reserves

Software reliability is still a problem (even in ultra-critical systems)

2005-2008:

- Malaysia Airlines Flight 124 (Boeing 777)

  "Software anomaly"

- Qantas Airlines Flight 72 (Airbus A330)

  Transient fault in the inertial reference unit

Software reliability is still a problem (even in ultra-critical systems)

|galois|

2005-2008:

- Malaysia Airlines Flight 124 (Boeing 777)
  - “Software anomaly”
- Qantas Airlines Flight 72 (Airbus A330)
  - Transient fault in the inertial reference unit
- Space Shuttle STS-124 aborted launch
  - Bad assumptions about distributed fault-tolerance

Lesson #4: Safe, Unified Host Language

- Embedded DSLs are a gestalt-shift for safety-critical languages
- Few front-end bugs: (no parser, lexer, etc.)
- Type-safe translation between DSLs in the same language

  E.g., seemless union of verification languages and programming languages
- The macro language is a build system, too!

```
compile program node
  (setCode (Just header)) baseOpts
```

```
distCompile program node headers =
  compile (program node) node
    (setCode (Just (headers node))) baseOpts
```

3 themes and a case-study

- RV for ultra-critical systems
  - Distributed systems
  - Hard real-time systems
  - Monitor hardware and software faults
- Using functional languages for monitor generation

  embedded domain-specific languages (eDSL)
- Low-cost, high assurance
- Case-study: aircraft guidance systems

Runtime verification is needed!

How do you know your embedded software won't fail?

- Certification (e.g., DO-178B) is largely process-oriented
- Testing exercises a small fraction of the state-space
- It's probably not formally verified
  - Even if so, just a small subsystem
  - And making simplifying assumptions

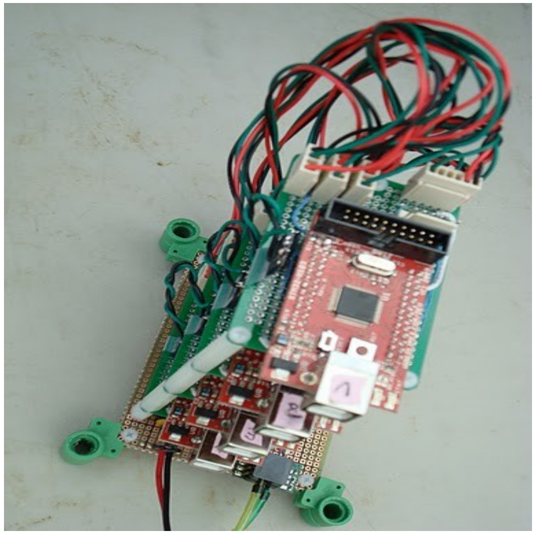
I'll argue: need the ability to detect/respond at runtime

Copilot Interpreter

```
evalExpr_ e0 exts locs strms = case e0 of
  Const _ x              -> x `seq` repeat x
  Drop t i id            -> strictList $
    let Just xs = lookup id strms >>= fromDynF t
    in  P.drop (fromIntegral i) xs
  Local t1 _  name e1 e2 -> strictList $
    let xs    = evalExpr_ e1 exts locs strms
        locs' = (name, toDynF t1 xs) : locs
    in  evalExpr_ e2 exts locs' strms
  Var t name             -> strictList $
    let Just xs = lookup name locs >>= fromDynF t in xs
  ExternVar t name       -> strictList $ evalExtern t name exts
  Op1 op e1              -> strictList $ repeat (evalOp1 op)
                                 <*> evalExpr_ e1 exts locs strms
  Op2 op e1 e2           -> strictList $ repeat (evalOp2 op)
                                 <*> evalExpr_ e1 exts locs strms
                                 <*> evalExpr_ e2 exts locs strms
  Op3 op e1 e2 e3        -> strictList $ repeat (evalOp3 op)
                                 <*> evalExpr_ e1 exts locs strms
                                 <*> evalExpr_ e2 exts locs strms
                                 <*> evalExpr_ e3 exts locs strms
```

Flight Tests

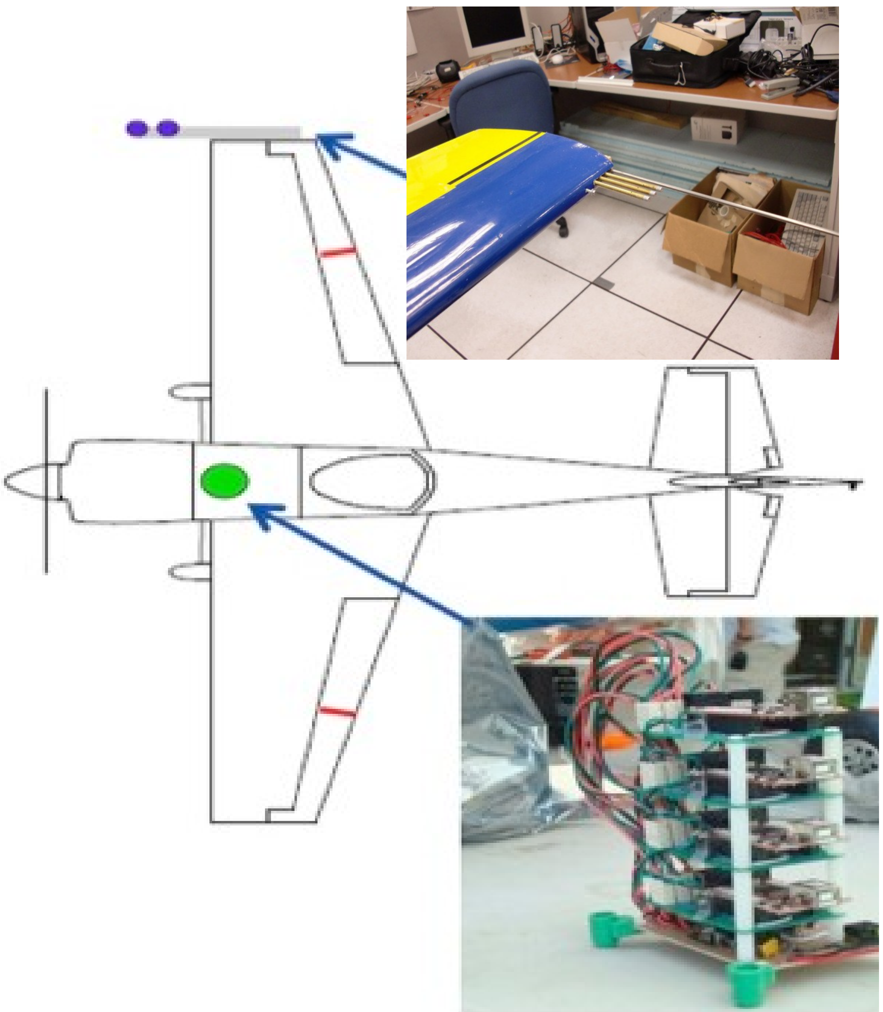
Experiment goals



- Monitors to check a distributed airspeed system
- Monitors also distributed & real-time

  “Bolt-on” fault-tolerance
- While satisfy timing, certifiability, SWaP goals
- Inject both physical and software faults

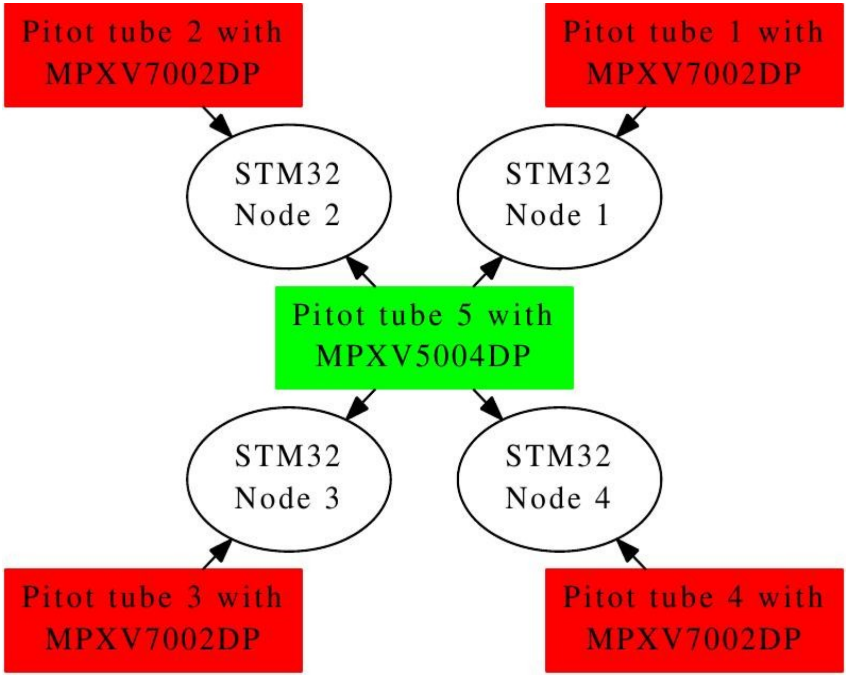
Aircraft configuration
Edge 540T



Monitoring experiments

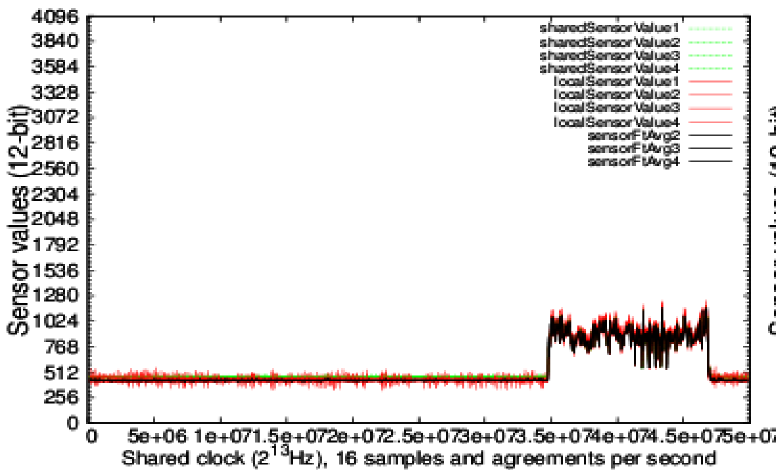
- Monitors communicate with one another over dedicated serial lines in real-time
- Properties
  - *Agreement*: return a fault-tolerant average of sensor values
    - Used to diagnose local faults
    - Diagnoses faults in the monitors **or** the sensor systems
  - Unrealistic sensor data

    Senors values change “too fast”
- Upshot: decomposable fault-tolerance

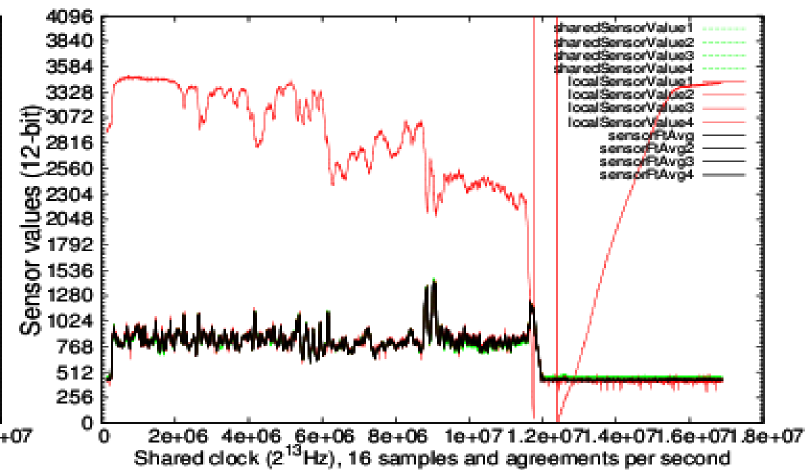


Monitoring results

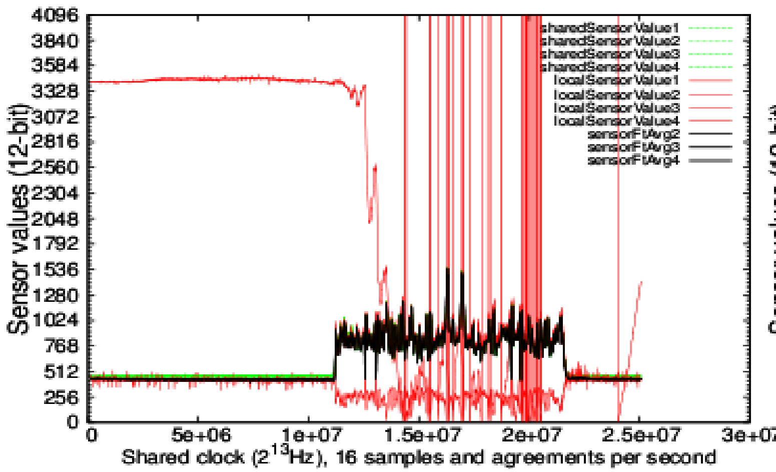
One Byzantine-faulty processor, plus



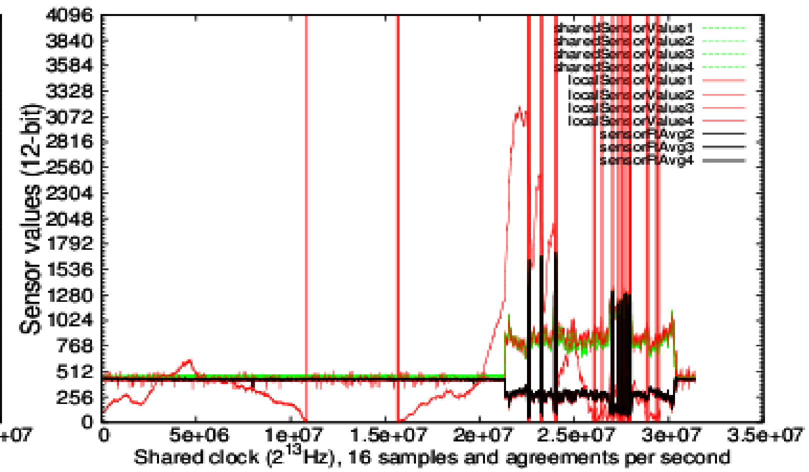
(c) All tubes unmodified



(d) One tube stuck



(e) Two tubes stuck



(f) Three tubes stuck

Future work

- Another case-study on autopilot communication system
- Tools for scheduling monitors
  - Used timer interrupts
  - And scheduler to decompose monitor's tasks (variable sampling, computation, etc.)
- Efficient compilation for eDSLs
- Automated mapping from real-time history to value history

  E.g., state in monitor that the Δ in *v* over 1sec. → monitor maintains a history buffer of x values.

Summary

- RV works and is needed for ultra-critical systems!
  - Distributed systems
  - Real-time systems
- Using functional languages for monitor generation

    eDSLs: "the benefits of functional languages applied to real-time embedded systems"
- Low-cost, high assurance

http://leepike.github.com/Copilot/

leepike/Copilot @ GitHub

http://leepike.github.com/Copilot/

Fork me on GitHub

Copilot

A (Haskell DSL) stream language for generating hard real-time C code.

Can you write a list in Haskell? Then you can write embedded C code using Copilot.
Here's a Copilot program that computes the Fibonacci sequence (over Word 64s) and tests for even numbers:

```
fib :: Streams
fib = do
  "fib" .= [0,1] ++ var "fib" + (drop 1 $ varW64 "fib")
  "t" .= even (var "fib")
    where even :: Spec Word64 -> Spec Bool
          even w = w `mod` const 2 == const 0
```

Copilot contains an interpreter, a compiler, and uses a model-checker to check the correctness of your program. The compiler generates constant time and constant space C code via Tom Hawkin's Atom Language (thanks Tom!). Copilot is specifically developed to write embedded software monitors for more complex embedded systems, but it can be used to develop a variety of functional-style embedded code.

Executing

```
> compile fib "fib" baseOpts
```

generates fib.c and fib.h (with a `main()` for simulation---other options change that). We can then run

```
> interpret fib 100 baseOpts
```

to check that the Copilot program does what we expect. Finally, if we have CBMC installed, we can run

```
> verify "fib.c"
```

to prove a bunch of memory safety properties of the generated program.

Differences From Lustre

- eDSL approach
- Polymorphic (embedded in Haskell)
- Simpler clock calculus—no projection operator
- BSD3
- V&V tools

Cheap assurance

Who watches the watchmen?

- Types are free proofs—use a typed language
- Reuse existing compiler infrastructure
- Automated random testing

    Ensure interpreter == compiler, millions of times

- Test coverage (line, branch, functional call) using *gcov*
- Automated back-end equivalence proofs (CBMC)

And it's all cheap & easy.

Air Data Inertial Reference Units
35+ years of failures

Failures cited in

- Northwest Orient Airlines Flight 6231 (1974)---3 killed

  Increased climb/speed until uncontrollable stall
- Birgenair Flight 301, Boeing 757 (1996)---189 killed

  One of three pitot tubes blocked; faulty air speed indicator
- Aeroperú Flight 603, Boeing 757 (1996)---70 killed

  Tape left on the static port(!) gave erratic data
- Líneas Aèreas Flight 2553, Douglas DC-9 (1997)---74 killed
  - Freezing caused spurious low reading, compounded with a failed alarm system
  - Speed increased beyond the plane's capabilities
- Qantas Flight 72 , Airbus A330---115 injuries
  - ADIRU failure, software "limitation"
- Air France Flight 447, Airbus A330 (2009)---228 killed
  - Airspeed "unclear" to pilots
  - Still under investigation

The power of eDSLs

- Some problems for conventional compilers go away
  - New language features are host-language macros
  - Don't need scripting languages
- E.g., compiling distributed monitors is just another host-language function:

```
compile program node
  (setCode (Just header)) baseOpts
```

```
distCompile program node headers =
  compile (program node) node
    (setCode (Just (headers node))) baseOpts
```