

Monitor Synthesis for Software Health Management

Lee Pike | leepike@galois.com
Galois, Inc.

Alwyn Goodloe | NASA Langley Research Center
Robin Morisset | Ecole Normale Supérieure
Sebastian Niller | National Institute of Aerospace
Nis Wegmann | Technical University of Denmark

2011 Annual Technical Meeting
May 10–12, 2011
St. Louis, MO

Need

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 to detect/respond at runtime

Yes, it's *Still* a Problem

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



Just the FaCTS, Ma'am: The Constraints



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

- **F**unctionality: don't change the target's behavior
- **C**ertifiability: don't require re-certification, or make it 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

**How do we monitor a system without
violating these constraints?**

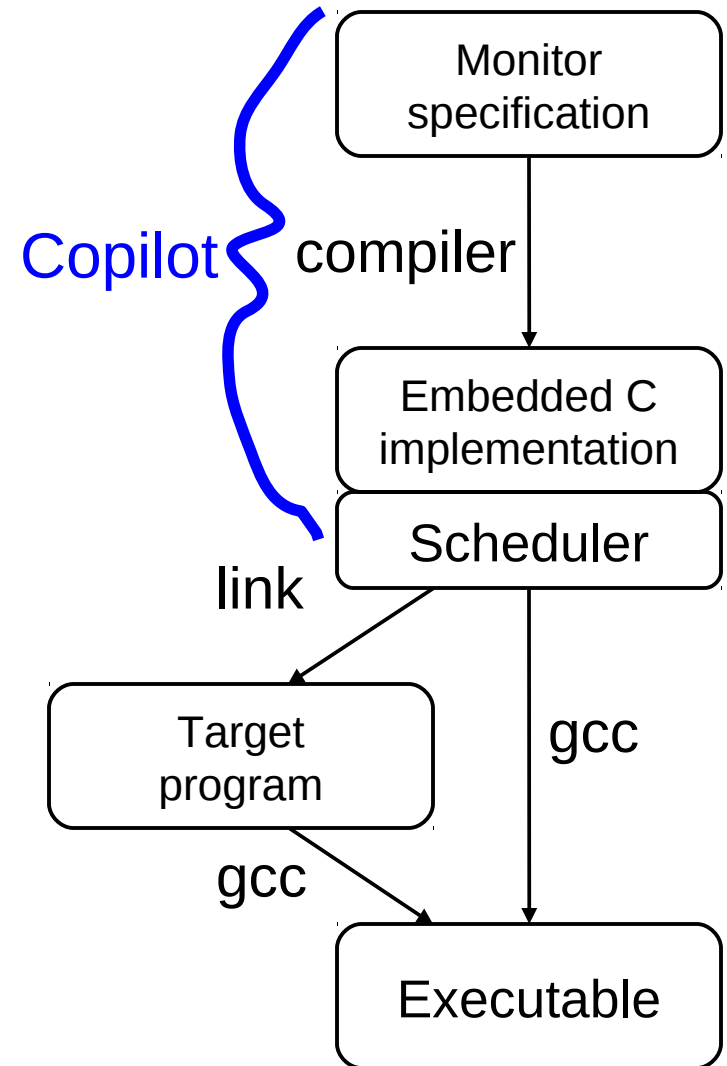
Outline

1. The *Copilot* language and compiler
2. Embedded domain-specific languages
3. Low-cost high-assurance
4. Pilot-study¹: injecting software faults in a fault-tolerant air-speed system
5. Conclusions

¹Pun intended

Copilot: Embedded System Monitoring

- *Copilot* is a language, compiler, and verification tools
- Compiles monitor specifications to embedded C
 - Constant time, constant space
 - Generates its own scheduler: no OS needed
- Time-triggered monitoring
- Monitor program:
 - **Inputs:** monitored memory
 - **Outputs:** trigger functions, if a monitor is violated



Copilot Language

A simple stream language

- Think: data-flow of infinite lists (streams) – LUSTRE
- Streams give a discrete, synchronous view of real-time
- Strongly & statically typed variables with no lossy casts

```
let x = varW64 "x"  
x      .= [0] ++ x + 2  
-----  
x → 0, 2, 4, 6 ...
```

Copilot Language

```
let x = varW64 "x"  
x .=[0] ++ x + 2
```

compile spec "foo"
baseOpts

```
void foo() {  
  {  
    static uint8_t __scheduling_clock = 1;  
    if (__scheduling_clock == 0) {  
      __r0(); /* foo.update__x */  
      __scheduling_clock = 4;  
    }  
    else  
      __scheduling_clock = __scheduling_  
  }  
  {  
    static uint8_t __scheduling_clock =  
    if (__scheduling_clock == 0) {  
      __r1(); /* foo.output__x */  
      . . .  
    }  
  }  
}
```

```
/* foo.update__x */  
static void __r0() {  
  bool __0 = true;  
  uint32_t __1 = 0UL;  
  uint32_t __2 =  
    copilotStatefoo.foo.outputIndex__x;  
  uint32_t __3 = __1 + __2;  
  uint32_t __4 = 2UL;  
  . . .  
}
```

State-machine functions

Schedule

Copilot Interpreter (In One Slide)

```
interpret copilotVs extVs s =  
  case s of  
    Const c -> repeat c  
    Var v -> getElem v copilotVs  
    ExtVar _ v -> checkV v (\v' -> (getElem v' extVs))  
    ExtArr _ (v,s') -> checkV v (\v' -> map (\i -> getElem v' extVs  
                                              !! fromIntegral i)  
                                              (interpret copilotVs extVs s'))  
    Append ls s' -> ls ++ interpret copilotVs extVs s'  
    Drop i s' -> drop i $ interpret copilotVs extVs s'  
    F f _ s' -> map f (interpret copilotVs extVs s')  
    F2 f _ s0 s1 -> zipWith f (interpret copilotVs extVs s0)  
                               (interpret copilotVs extVs s1)  
    F3 f _ s0 s1 s2 -> zipWith3 f  
                        (interpret copilotVs extVs s0)  
                        (interpret copilotVs extVs s1)  
                        (interpret copilotVs extVs s2)
```

Point #1: Embedded DSLs

Make Things Better

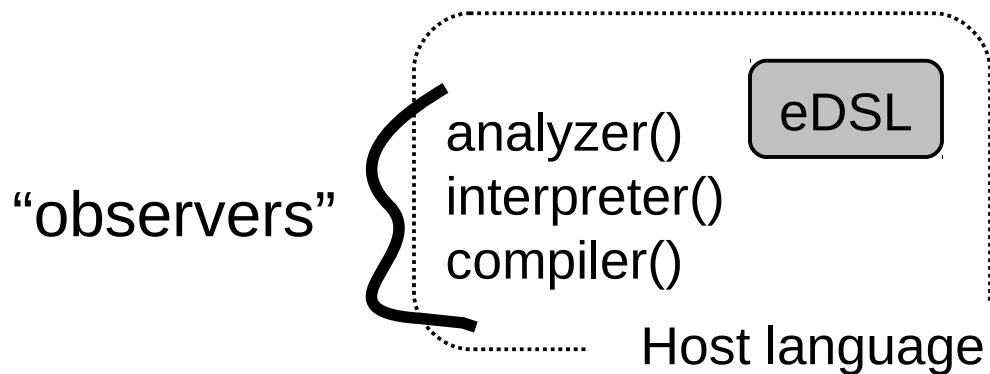


- A **domain-specific language (DSL)** is a special-purpose programming language.

E.g., sed/awk, Simulink, R

- An **embedded DSL (eDSL)** is a DSL written as a library in a general-purpose programming language

Often the host language is a functional language, e.g., Haskell, Scheme, OCaml



Point #1: Embedded DSLs

Make Things Better

Why eDSLs?

- Lexer, parser, type-checker, etc. for free and more likely correct
- Macro language for free (the entire host language)

In eDSLs, the macro language is primary


- Libraries for free
- Much easier to make your own modifications

For Copilot: can we have the advantages of functional languages without suffering its limitations (timing, control-flow, memory size)?

Point #1: Embedded DSLs (Sometimes!) Make Things Better



Why not?

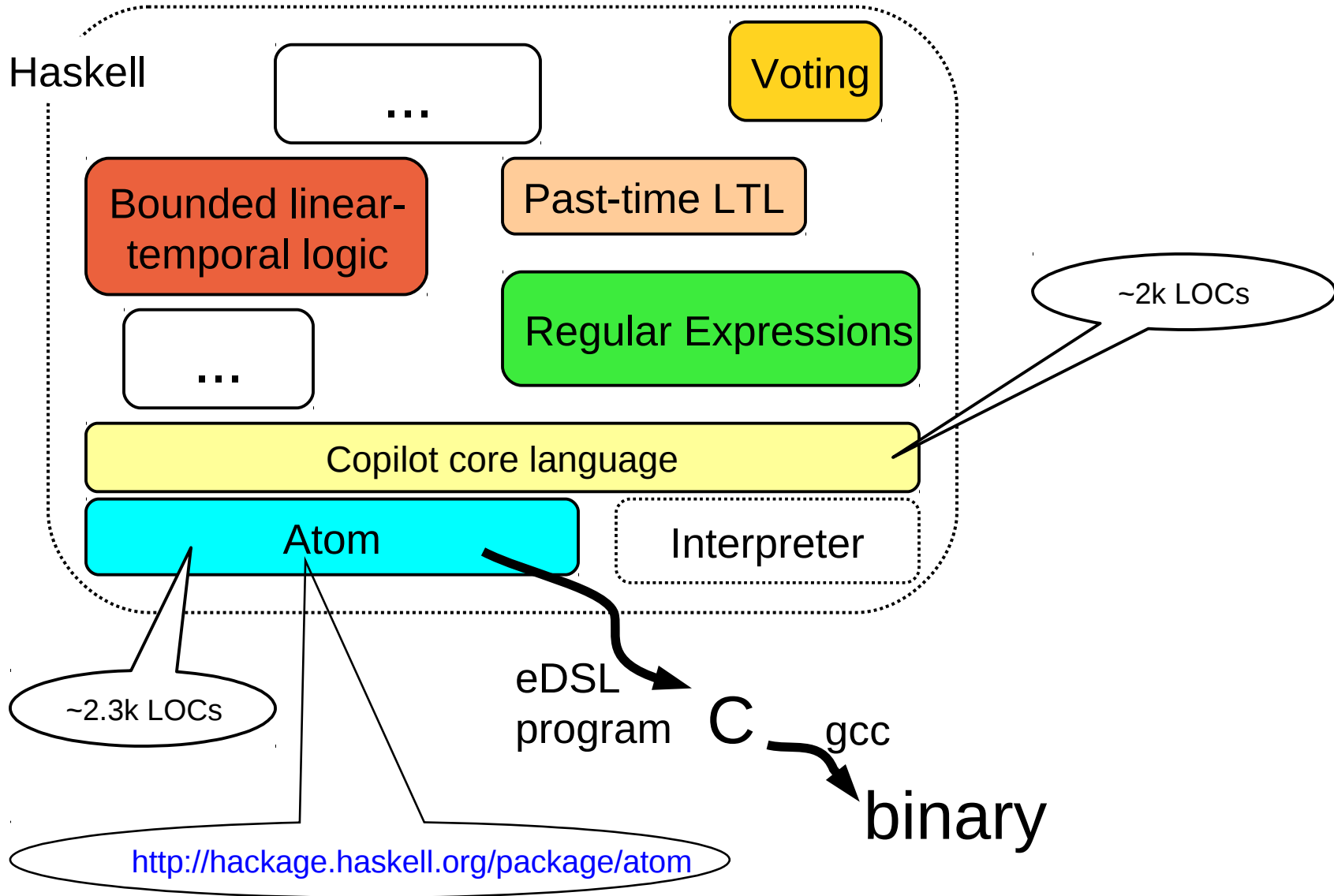
- 
- A large blue curly bracket on the left side of the list, grouping the first four items.
- The DSL syntax must be a “sub-syntax” of your host language
 - In some cases, efficiency can be tricky
 - More esoteric error messages
 - eDSLs in certification unexplored
 - Harder to make proprietary/closed source

Research
topics!

eDSLs: C'mon, Everybody's Doing It

- Eaton (embedded control systems)
- Ericsson (DSP)
- Credit Suisse and other trading houses (e.g., derivatives pricing)
- Galois (Numerous)

Copilot as an eDSL



The Power of eDSLs

```
-- external variables
t0      = extW8 "temp_probe_0"
t1      = extW8 "temp_probe_1"
t2      = extW8 "temp_probe_2"
cooler  = extB  "fan_status"
-- Copilot variables
maj      = varW8 "maj"
check    = varB  "maj_check"
overHeat = varB  "over_heat"
monitor  = varB  "monitor"

-----
engineMonitor = do
  let temps = map (< 250) [t0, t1, t2]
  maj      .= majority temps
  check    .= aMajority temps maj
  overHeat `ptltl`
    ((cooler || maj && check)
     `since` not maj)
  monitor .= not overHeat
  trigger monitor "shutoff" void
```

Libraries

"If the majority of the three engine temperature probes has exceeded 250 degrees, then the cooler is engaged and remains engaged until the temperature of the majority of the probes drop to 250 degrees or less. Otherwise, trigger an immediate shutdown of the engine."

approx.
800LoCs of C

Point #2: Low-Cost High-Assurance

Who watches the watchmen?

Some lessons:

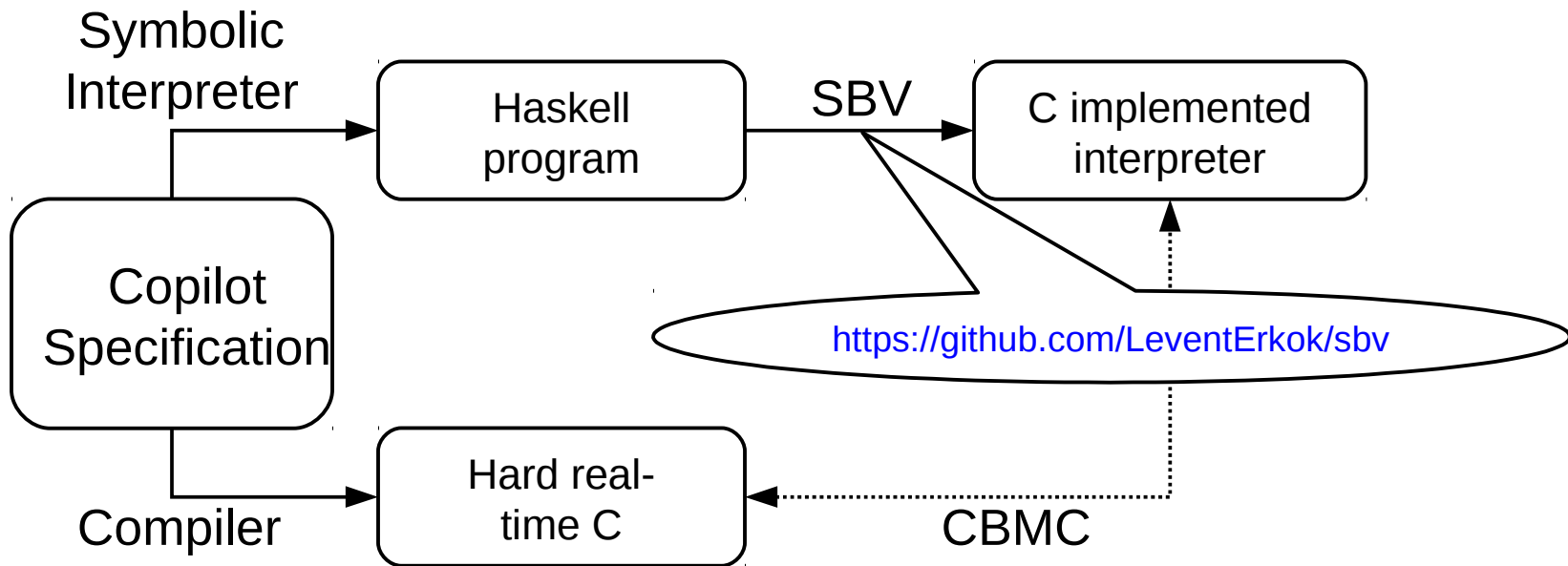
- Types are free proofs
- (Try) to avoid compiler bugs/non-standard behavior
- Compile -Wall, compile -Wall, compile -Wall
- Ensure interpreter == compiler
- Ensure interpreter == compiler, millions of times
- Test coverage (line, branch, functional call) using *gcov*

Point #2: Low-Cost High-Assurance

- Prove memory-safety.

CBMC <http://www.cprover.org/cbmc/>

- Verify the compilation – a “poor man's verifying compiler”
(future work)



Interlude: Pitot Failures



WIKIPEDIA
The Free Encyclopedia

Article Discussion

Air Data Inertial Reference Unit

An **Air Data Inertial Reference Unit** (ADIRU) is a key compon

Failures and directives

FAA Airworthiness directive 2000-07-27

On May 3, 2000, the FAA issued airworthiness directive 2000-07-27, address Boeing 737, 757, Airbus A319, A320, A321, A330, and A340 models.^{[2][10]}

Airworthiness directive 2003-26-03

On 27 January 2004 the FAA issued airworthiness directive 2003-26-03 (lat

Alitalia A-320

On 25 June 2005, an [Alitalia Airbus A320-200](#) registered as I-BIKE departed failed, leaving only one operable. In the subsequent confusion the third was

Malaysia Airlines Flight 124

On 1 August 2005 a serious incident involving [Malaysia Airlines Flight 124](#), aircraft acting on false indications.^[14] In that incident the incorrect data imp with the stall warning activated. The pilots recovered the aircraft with the ai

Aviation Today
Your First Destination For Global Industry Intelligence

Home | Avionics | Rotor & Wing | Air Safety Week | Aircraft Value News

View by Category: Military | Commercial | Business & General Aviation | Rotorcraft | Air Traffic C

SEARCH

Monday, February 7, 2011

More Pitot Tube Incidents Revealed

New reports of Pitot tube malfunctions on [Airbus](#) jetliners during prompted additional safety concerns about their reliability.

Reliability of the air pressure sensors made by both Thales and Goodr particular prominence after the 2009 crash of an Air France [Airbus](#) A330-300 from Rio de Janeiro to Paris that claimed 228 victims. Pitot Tubes are essen

AVIALL

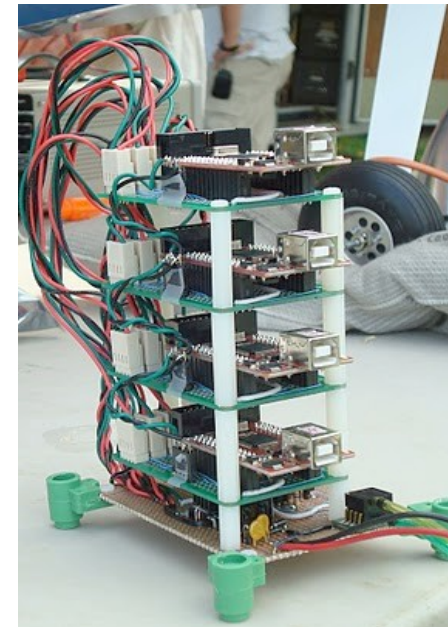
Interlude: Pitot Failures

Failures cited in

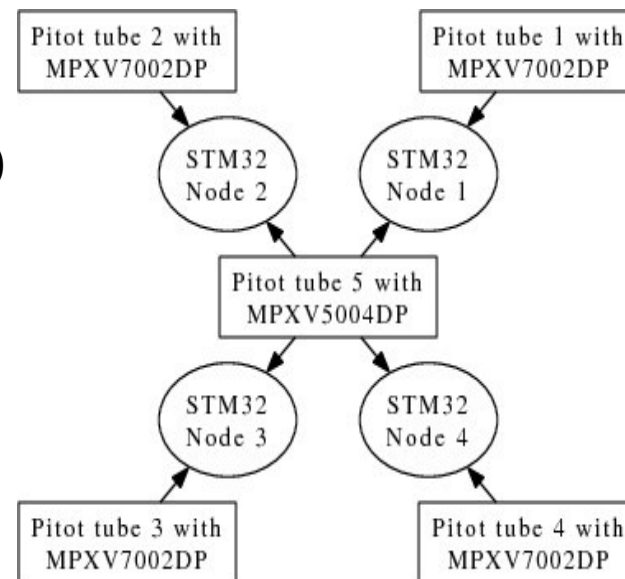
- Northwest Orient Airlines Flight 6231 (1974)---3 killed
Increased climb/speed until uncontrollable stall
- Birgenair Flight 301, Boeing 757 (1996)---189 killed
One of three pitot tubes blocked; faulty air speed indicator
- Aeroperú Flight 603, Boeing 757 (1996)---70 killed
Tape left on the static port(!) 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
- Air France Flight 447, Airbus A330 (2009)---228 killed
 - Airspeed “unclear” to pilots
 - Still under investigation
- ...

Test Bed

- Representative of fault-tolerant systems
- 4 X STM32 microcontrollers
- ARM Cortex M3 cores clocked at 72 Mhz
- 5 differential pressure sensors
 - Senses dynamic and static pitot tube pressure
 - Pitot tubes measure airspeed
- Designed to fit UAS (unpiloted air system)
 - Size, power, weight,...

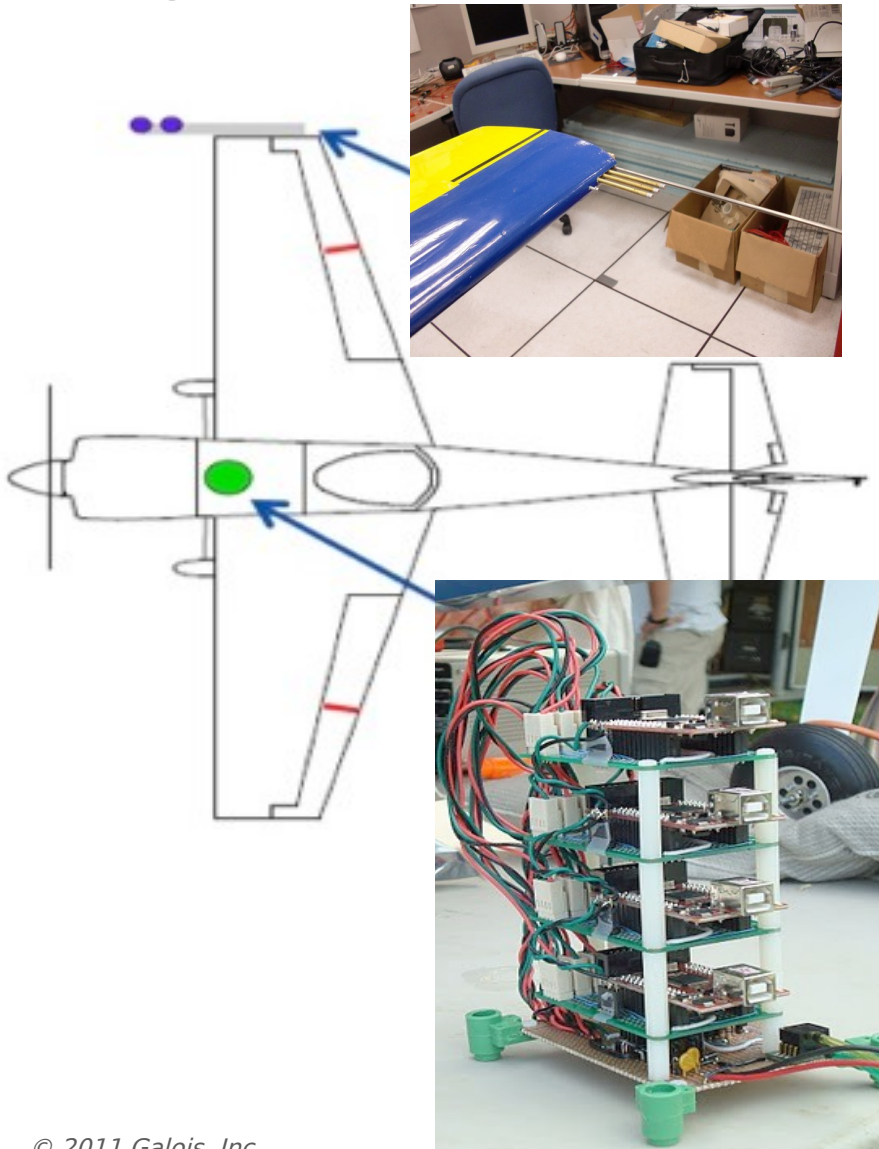


|galois|



Aircraft Configuration

Edge 540T-R2



Copilot Monitors

Introduced **software faults** to be caught by Copilot monitors:

- Abrupt airspeed change: airspeed $\Delta > 12$ m/s
- Fault-management assumptions
 - Fault-management used the Boyer-Moore majority vote algorithm
 - Check agreement between the voted values
 - Uses coordinating distributed monitors
- Subsequent flights:
 - Ground-station communication protocol
 - Other sensors

Monitoring Results


- Monitoring approach did not disrupt the FaCTS properties of the observed system
 - Under ~100 C expressions per monitor
 - Binaries on the order of 10k
- Monitoring via sampling works for periodic tasks
- Simulated mode change
- *Next time*: didn't think to monitor for a taped pitot tube!



Future Work Test-Bed

In collaboration with Portland State University

- ArduPilot autopilot (open source)
- Altitude hold (barometer & sonar)
- Position hold (GPS magnetometer)
- Collision avoidance (infrared)
- Stabilization (gyroscope)
- Battery monitoring

approx. 
\$400 parts



Download, Develop, Use

|galois|

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

BSD3

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.

Future Work

- The steering problem (mode change)
Right now: escape to raw C
- Timing analysis: to monitor property p , need to sample at rate r
E.g., state-based properties
- Security monitoring for embedded systems
Tech transfer to AFRL

Conclusions

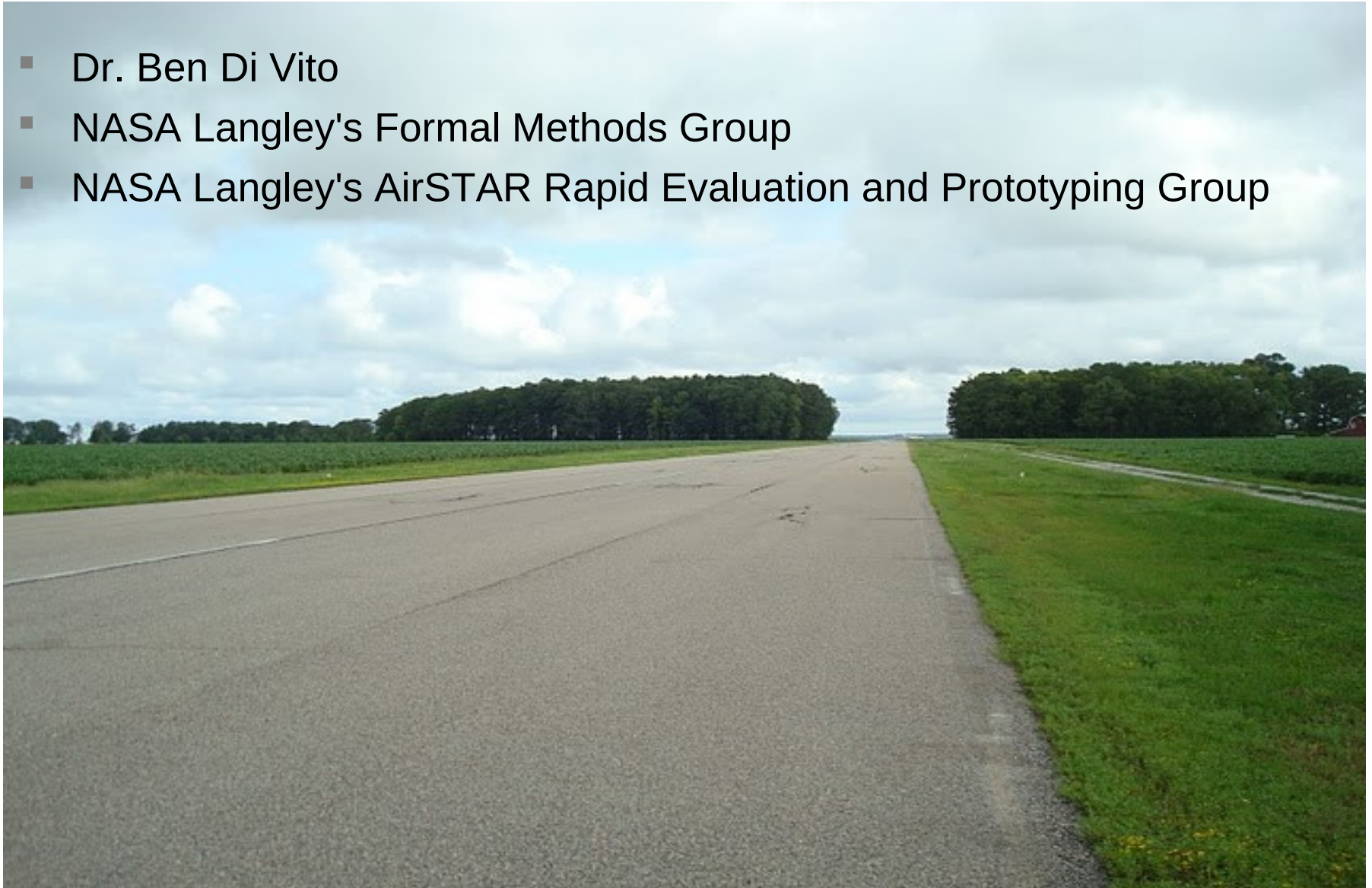
- Problem space: hard real-time embedded C
 - The FaCTS: Functionality, Certifiability, Timing, SWaP
 - Approach: monitoring by periodic sampling

- The eDSL approach
 - A path to fast, reliable compilers and languages

- *Nobody* watches the watchmen
 - Prove/test/verify your compiler is correct

Thanks

- Dr. Ben Di Vito
- NASA Langley's Formal Methods Group
- NASA Langley's AirSTAR Rapid Evaluation and Prototyping Group



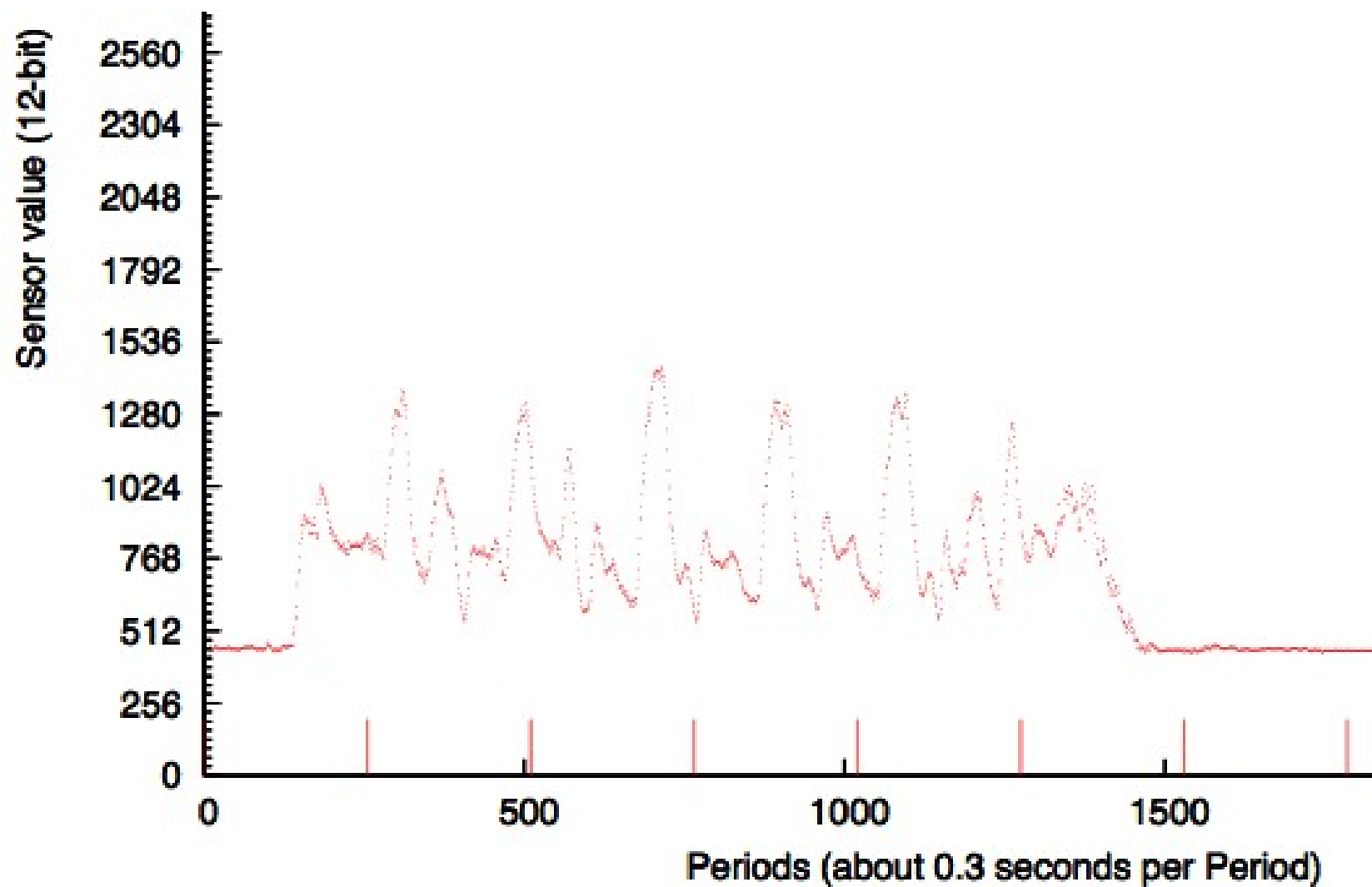
Appendix

Monitoring By Sampling

Without inlining monitors, we must sample:

- Property (011)*
- False positive (monitor misses an fault):
 - Values are 011**1**011 but sampling 011011
- False negative (monitor signals a fault that didn't occur):
 - Values are 011011 but sampling 011**1**011
- Observation: with fixed periodic schedule and shared clock
 - False negatives impossible
 - We don't want to re-steer an unbroken system
 - False positives possible, but requires constrained misbehavior

Pitot Data



- Gui
- --> Lustre
- Scheduling on ARINC 653
- Rushby: Liam(sp? flight) the control sampling/smoothing data
- Overflow vars monitoring
- level C system level A monitor -- DO178B

Stream Semantics (Append)

all operators are
lifted in Copilot

```
let x = varW64 in
```

```
    x .= [0, 1, 2] ++ x + 3
```

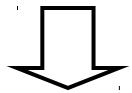
```
f [0, 1, 2]
```

```
  where f :: [Word64] -> [Word64]
```

```
        f x = x ++ f (map (+3) x)
```

```
x = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ...
```

```
x = [0, 1, 2]
```



(+3)

```
[3, 4, 5]
```



(+3)

```
[6, 7, 8]
```

...

(Copilot)

(Haskell)

Timed Semantics

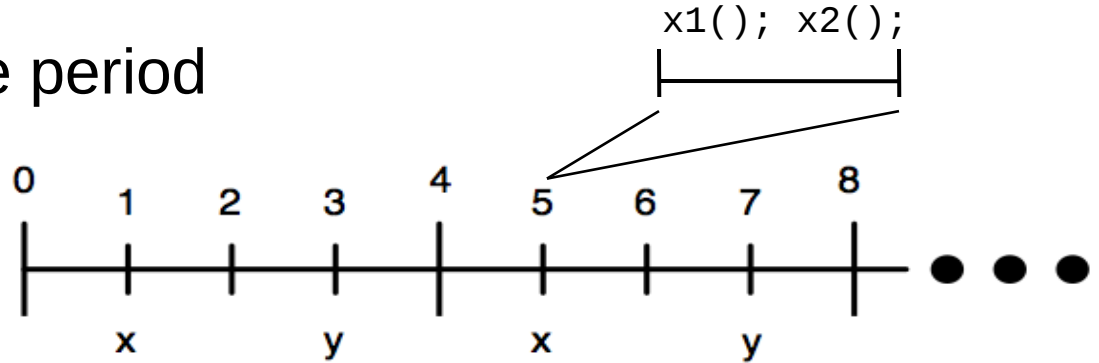
▪ **Period:** duration between discrete events

▪ **Phase:** offsets into the period

▪ **Example:**

▪ x: period 4 phase 1

▪ y: period 4 phase 3



▪ **Copilot ensures synchronization between streams**

▪ Assuming synchronization of phases in distributed systems: no non-faulty processor reaches the start of phase $p+1$ until every non-faulty processor has started phase p

Timed Semantics

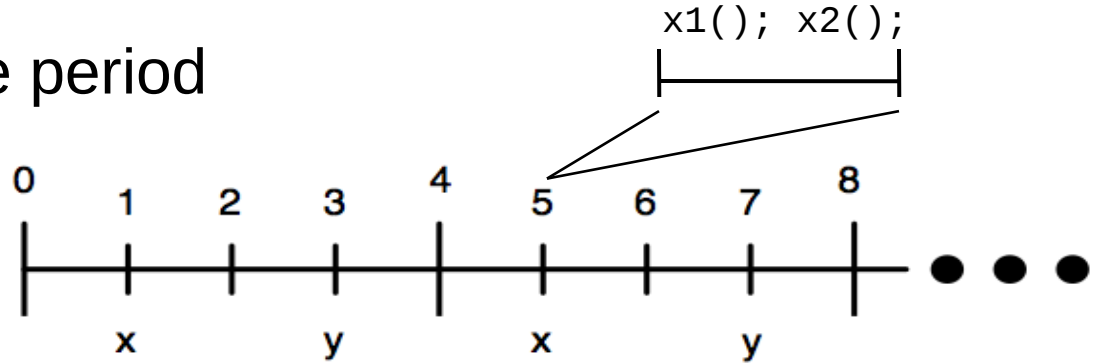
▪ **Period:** duration between discrete events

▪ **Phase:** offsets into the period

▪ **Example:**

▪ x: period 4 phase 1

▪ y: period 4 phase 3



▪ **Copilot ensures synchronization between streams**

▪ Assuming synchronization of phases in distributed systems: no non-faulty processor reaches the start of phase $p+1$ until every non-faulty processor has started phase p

Stream Semantics (Append)

all operators are
lifted in Copilot

```
let x = varW64 in
```

```
    x .= [0, 1, 2] ++ x + 3
```

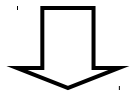
```
f [0, 1, 2]
```

```
    where f :: [Word64] -> [Word64]
```

```
          f x = x ++ f (map (+3) x)
```

```
x = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ...
```

```
x = [0, 1, 2]
```



(+3)

```
[3, 4, 5]
```



(+3)

```
[6, 7, 8]
```

...

(Copilot)

(Haskell)

Stream Semantics (Drop)



```
x .= [0, 1, 2] ++ x + 3
```

```
y .= drop 2 x
```

```
x = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ...
```

```
y = 2, 3, 4, 5, 6, 7, 8, 9, 10 ...
```

Stream Semantics (Drop)



```
x .= [0, 1, 2] ++ x + 3
```

```
y .= drop 2 x
```

```
x = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ...
```

```
y = 2, 3, 4, 5, 6, 7, 8, 9, 10 ...
```

Sample Code Generated (Incomplete)

state-update function
for *trigger* stream

```
engine :: Streams
engine = do
  ...
  trigger = (var overTempRise)
            ==> (extB shutoff 2)
```

external variable
sample function

```
/* engine.sample__shutoff_2 */
static void __r6() {
  bool __0 = true;
  bool __1 = shutoff;
  if (__0) {
  }
  engine.tmpSampleVal__shutoff_2 = __1;
}
```

```
/* engine.updateOutput__trigger */
static void __r0() {
  bool __0 = true;
  bool __1 = engine.tmpSampleVal__shutoff_2;
  bool __2 = ! __1;
  float __3 = 2.3F;
  uint64_t __4 = 0ULL;
  uint64_t __5 = engine.outputIndex__temps;
  uint64_t __6 = __4 + __5;
  uint64_t __7 = 4ULL;
  uint64_t __8 = __6 % __7;
  float __9 = engine.prophVal__temps[__8];
  float __10 = __3 + __9;
  uint64_t __11 = 2ULL;
  uint64_t __12 = __11 + __5;
  uint64_t __13 = __12 % __7;
  float __14 = engine.prophVal__temps[__13];
  bool __15 = __10 < __14;
  bool __16 = __2 && __15;
  bool __17 = ! __16;
  if (__0) {
  }
  engine.outputVal__trigger = __17;
}
```

Copilot Language Restrictions

Design goal: make memory usage constant and “obvious” to the programmer

- No anonymous streams
 - Compiler doesn’t have to worry about sharing
- No lazily-computed values
 - E.g. $x \text{ .} = [0] + x + 1$
 $y \text{ .} = \text{drop } 2 \ x$
- Other restrictions (see paper)
- Upshot: “WYSIWYG memory usage”
 - Memory constrained by number of streams
 - Memory for each stream is essentially the LHS of ++
 - Doesn’t include stack variables

Timing Info & Expression Counts

Timing
info

Period	Phase	Exprs	Rule
-----	-----	-----	----
3	0	18	engine.updateOutput__trigger
3	0	14	engine.updateOutput__overTempRise
3	0	3	engine.update__temps
3	1	7	engine.output__temps
3	1	2	engine.sample__temp_1
3	2	6	engine.incrUpdateIndex__temps
3	2	2	engine.sample__shutoff_2

		52	

Hierarchical Expression Count

helps with
WCET analysis

Expression
count

Total	Local	Rule
-----	-----	----
52	0	engine
6	6	incrUpdateIndex__temps
7	7	output__temps
2	2	sample__shutoff_2
2	2	sample__temp_1
14	14	updateOutput__overTempRise
18	18	updateOutput__trigger
3	3	update__temps

Generated engine.c and engine.h
Moving engine.c and engine.h to ./ ...
Calling the C compiler ...
gcc ./engine.c -o ./engine -Wall

Example Copilot Specification

“If the temperature rises more than 2.3 degrees within 2 seconds, then the engine has been shut off.” (period == 1 sec)

```
engine :: Streams
engine = do
  -- external vars
  let temp      = extF "temp" 1
  let shutoff   = extB "shutoff" 2
  -- Copilot vars
  let temps     = varF "temps"
  let overTemp  = varB "overTemp"
  let trigger   = varB "trigger"

  temps      .= [0, 0, 0] ++ temp
  overTemp   .= drop 2 temps > 2.3 + temps
  trigger    .= overTemp ==> hutoff
```

phases to
sample in

initial “don’t care”
values

Usage

- `compile spec "c-name" [opts] baseOpts`
- `interpret spec rounds [opts] baseOpts`
- `test rounds [opts] baseOpts`
 - quickChecking the compiler/interpreter
- `verify filepath int`
 - SAT solving on the generated C program
- `help (commands and options)`
- `[spec] (parser)`
- `Opts (incomplete list):`
 - C trigger functions
 - Ad-hoc C code (library included for writing this)
 - Hardware clock
 - Verbosity
 - GCC options

Runtime Monitoring: What's New?

- Not new:

- One-out-of-two systems
- Error-checking codes
- Distributed fault-tolerance
- Built-in test

Common source
of faults



- New(er) ideas:

- Domain-specific languages for monitoring
- High-assurance monitors
- **SW as a system component**

Decompose monitoring and controlling

Types

- Types: Int & Word (8, 16, 32, 64), Float, Double
- Each stream has a unique inferred type:

```
let x = varW64 "x"
x .= [0, 1] ++ x + 3
```

*inferred
types*

- Casting

- Implicit casting is a type-error
Won't compile

~~let x = varW64 "x"~~
~~let y = varW32 "y"~~
~~x .= y~~

- Explicit casting guarantees:
 - signs never lost (no Int --> Word casts)
 - No overflow (no cast to a smaller width)

```
let x = varB "x"
let y = varI32 "y"
x .= [True] ++ not x
y .= cast x + 4
```

The Power of eDSLs

Some problems for conventional compilers go away

- Don't have to add new language features (often)
- Don't need scripting languages

E.g., compiling distributed monitors is just another function:

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

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

Real-Time Runtime Monitoring

Lee Pike | **Galois, Inc.** | leepike@galois.com

joint work with

Alwyn Goodloe NASA Langley Research Center

Robin Morisset École Normale Supérieure

Sebastian Niller National Institute of Aerospace

Nis Wegmann Technical University of Denmark

