RV for Ultra-Critical Systems

Lee Pike  Galois, Inc. <leepike@galois.com>
Sebastian Niller  National Institute of Aerospace
Alwyn Goodloe  NASA Langley Research Center
Robin Morisset  École Normale Supérieure
Nis Wegmann  Technical University of Copenhagen

© 2011 Galois, Inc.
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
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
- Space Shuttle STS-124 aborted launch
  Bad assumptions about distributed fault-tolerance
Monitoring constraints

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

- **Functionality**: don’t change the target’s behavior
  
  No false positives!

- **Certifiability**: don’t require re-certification, or make it easy
  
  Don't go changing sources.

- **Timing**: don’t interfere with the target’s timing

- **SWaP**: don’t exhaust size, weight, power reserves

How do we monitor a system without violating these constraints?
Our answer

- Synthesize monitors
  - From high-level specs, generate purely functional C99 Lustre-like stream language → Purely functional Misra-like C
  - Hard real-time: easy to compute WCET
    - Scheduler to give fine-grained timing control
    - No RTOS needed

- *Time-triggered monitoring*:
  - Sample program variables periodically
  - Keep histories as needed
  - *Not* addressing control-flow
Sample Copilot specification

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.

```
engineMonitor = do
    trigger "shutoff" (not ok) [arg maj]

where

vals = map externW8 ["tmp_probe_0", "tmp_probe_1", "tmp_probe_2"]
exceed = map (< 250) vals
maj = majority exceed
checkMaj = aMajority exceed maj
ok = alwaysBeen ((maj && checkMaj) ==> extern "cooler")
```

Key: library functions trigger macros
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

© 2011 Galois, Inc.
Copilot architecture

Libraries

Copilot specification language

Core language

Interpreter

Hard real-time back-end + scheduler (C)

Hard real-time back-end (C)

... Kind, other code generators

LTL
ptLTL
Regular expressions
clocks
fault-tolerance
etc.

Type-checking,
causality analysis,
etc.

QuickCheck
testing

CBMC: (C bounded model-checker)
Copilot architecture

- **Core language**
- **Libraries**
- **Copilot specification language**
- **Haskell**
  - Type-checking, causality analysis, etc.
  - Embedded domain-specific language (eDSL)

- **Hard real-time back-end (C)**
- **Hard real-time back-end + scheduler (C)**
- **Kind, other code generators**
- **CBMC: (C bounded model-checker)**
- **QuickCheck testing**

- **Regular expressions clocks fault-tolerance etc.**
- **LTL ptLTL**
Flight Tests
Pitot tube failures

© 2011 Galois, Inc.
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
- Air France Flight 447, Airbus A330 (2009)---228 killed
  - Airspeed “unclear” to pilots
  - Still under investigation
- ...

© 2011 Galois, Inc.
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
  - Sensors 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 $\Delta$ in $v$ over 1sec. $\rightarrow$ 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
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:

```haskell
fib :: Streams
fib = do
  "fib" << \( [0,1] \) ++ var "fib" + (drop 1 $ varW64 "fib")
  "c" . even (var "fib")
  where even (var "fib") = 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 Hawkins’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

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

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

```haskell
> interpret fib 100 baseOpts
```

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

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

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

© 2011 Galois, Inc.
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.
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
```