Copilot: Monitoring Embedded Systems

Lee Pike and Nis Wegmann and Sebastian Niller and Alwyn Goodloe

the date of receipt and acceptance should be inserted later

Abstract Abstract Runtime verification (RV) is a natural fit for ultra-critical systems that require correct software behavior. Due to the low reliability of commodity hardware and the adversity of operational environments, it is common in ultra-critical systems to replicate processing units (and their hosted software) and incorporate fault-tolerant algorithms to compare the outputs, even if the software is considered to be fault-free. In this paper we investigate the use of software monitoring in distributed fault-tolerant systems and the implementation of fault-tolerance mechanisms using RV techniques, we describe the Copilot language and compiler that generates monitors for distributed real-time systems, and we discuss two case-studies in which Copilot-generated monitors were used to detect onboard software and hardware faults and monitor air-ground data link messaging protocols.

1 Introduction

One in a billion, or 10^{-9}, is the prescribed safety margin of a catastrophic fault occurring in the avionics of a civil aircraft [29]. The justification for the requirement

Lee Pike
Galois, Inc., Portland, Oregon
E-mail: leepike@galois.com

Nis Wegmann
University of Copenhagen, Copenhagen, Denmark
E-mail: wegmann@diku.dk

Sebastian Niller
Unaffiliated
E-mail: sebastian.niller@gmail.com

Alwyn Goodloe
NASA Langley Research Center, Hampton, Virginia
E-mail: a.goodloe@nasa.gov

is to show that failures resulting in a catastrophic effect are "so unlikely that it is not anticipated to occur during the operational life of an entire system or fleet"[1]. Let us call systems with reliability requirements on this order *ultra-critical* and those that meet the requirements *ultra-reliable*. Similar reliability metrics might be claimed for other safety-critical systems, like nuclear reactor shutdown systems or railway switching systems.

Neither formal verification nor testing can ensure system reliability. Contemporary ultra-critical systems may contain millions of lines of code; the full functional correctness of approximately ten thousand lines of code represents the state-of-the-art, taking 20 engineer-years to achieve [18]. Nearly 20 years ago, Butler and Finelli showed that testing alone cannot verify the reliability of ultra-critical software [7].

Runtime verification (RV), where monitors detect and respond to property violations at runtime, holds particular potential for ensuring that ultra-critical systems are in fact ultra-reliable, but there are challenges. In ultra-critical systems, RV must account for both software and hardware faults. Whereas software faults are design errors, hardware faults can be the result of random failure.

For the purposes of this paper, assume that characterizing a system as being *ultra-critical* implies it is a distributed system with replicated hardware (so that the failure of an individual component does not cause system-wide failure); also assume ultra-critical systems are embedded systems responsible for sensing and/or controlling some physical plant and that they are *hard real-time*, meaning that deadlines are fixed and time-critical.

Outline. In Section 2, we motivate the need for runtime verification for ultra-critical systems. In Section 3,

we describe constraints and approaches to RV for embedded systems. In Section 4, we present the language of Copilot, our approach for ultra-critical RV. We describe the tools (e.g., the interpreter and compiler backends) associated with Copilot in Section 5. We present two case-studies applying Copilot to avionic systems in Section 6. Final remarks are given in Section 7.

2 When Ultra-Critical Is *Not* Ultra-Reliable

Well-known, albeit dated, examples of the failure of critical systems include the Therac-25 medical radiation therapy machine [22] and the Ariane 5 Flight 501 disaster [25]. However, more recent events show that critical-system software safety, despite certification and extensive testing, is still an unmet goal. Below, we briefly overview three examples drawn from faults in the Space Shuttle, a Boeing 777, and an Airbus A330, all occurring between 2005 and 2008.

Space Shuttle. During the launch of shuttle flight Space Transportation System 124 (STS-124) on May 31, 2008, there was a pre-launch failure of the fault diagnosis software due to a "non-universal I/O error" in the Flight Aft (FA) multiplexer de-multiplexer (MDM) located in the orbiter's aft avionics bay [4]. The Space Shuttle's data processing system has four general purpose computers (GPC) that operate in a redundant set. There are also twenty-three MDM units aboard the orbiter, sixteen of which are directly connected to the GPCs via shared buses. The GPCs execute redundancy management algorithms that include a fault detection, isolation, and recovery function. In short, a diode failed on the serial multiplexer interface adapter of the FA MDM. This failure was manifested as a *Byzantine fault* (i.e., a fault in which different nodes interpret a single broadcast message differently [20]), which was not tolerated and forced an emergency launch abortion.

Boeing 777. On August 1, 2005, a Boeing 777-120 operated as Malaysia Airlines Flight 124 departed Perth, Australia for Kuala Lumpur, Malaysia. Shortly after takeoff, the aircraft experienced an in-flight upset, causing the autopilot to dramatically manipulate the aircraft's pitch and airspeed. A subsequent analysis reported that the problem stemmed from a bug in the Air Data Inertial Reference Unit (ADIRU) software [6]. Previously, an accelerometer (call it A) had failed, causing the fault-tolerance computer to take data from a backup accelerometer (call it B). However, when the backup accelerometer failed, the system reverted to taking data from A. The problem was that the fault-tolerance software assumed there would not be a simultaneous failure of both accelerometers. Moreover, the software was not designed to report accelerometer A's failure so maintenance was not performed.

Airbus A330. On October 7, 2008, an Airbus A330 operated as Qantas Flight QF72 from Singapore to Perth, Australia was cruising when the autopilot caused a pitch-down followed by a loss of altitude of about 200 meters in 20 seconds (a subsequent less severe pitch was also made) [23]. The accident required the hospitalization of fourteen people. Like in the Boeing 777 upset, the source of this accident was an ADIRU. The ADIRU appears to have suffered a transient fault that was not detected by the fault-management software of the autopilot system.

3 Runtime Monitoring for Embedded Systems: Constraints and Approaches

In this section, we present constraints to runtime monitoring for real-time embedded systems.

3.1 RV Constraints

Ideally, the RV approaches that have been developed in the literature could be applied straightforwardly to ultra-critical systems. Unfortunately, typical RV approaches violate constraints imposed on ultra-critical systems. We summarize these constraints using the acronym "FaCTS":

- **F**unctionality: the RV system cannot change the target's behavior (unless the target has violated a specification).
- **C**ertifiability: the RV system must not make re-certification (e.g., DO-178B [28]) of the target onerous.
- **T**iming: the RV system must not interfere with the target's timing.
- **S**WaP: The RV system must not exhaust size, weight, and power (SWaP) tolerances.

The functionality constraint is common to all RV systems, and we will not discuss it further. The certifiability constraint is at odds with aspect-oriented programming techniques, in which source code instrumentation occurs across the code base—an approach classically taken in RV (e.g., the Monitor and Checking (MaC) [17] and Monitor Oriented Programming (MOP) [8] frameworks). For codes that are certified, instrumentation is not a feasible approach, since it requires costly reevaluation of the code. Source code instrumentation can modify both the control flow of the instrumented program as well as its timing properties. Rather, an

RV approach must isolate monitors in the sense of minimizing or eliminating the effects of monitoring on the observed program's control flow.

Timing isolation is also necessary for real-time systems to ensure that timing constraints are not violated by the introduction of RV. Assuming a fixed upper bound on the execution time of RV, a worst-case execution-time analysis is used to determine the exact timing effects of RV on the system—doing so is imperative for hard real-time systems.

Code and timing isolation requirements cause the most significant deviations from traditional RV approaches. Section 3.2 argues that these requirements dictate a *time-triggered* RV approach, in which a program's state is periodically sampled based on the passage of time rather than occurrence of events [26].

The final constraint, SWaP, applies both to memory (embedded processors may have just a kilobytes of available memory) as well as additional hardware (e.g., processors or interconnects).

3.2 Timed-Triggered Monitoring

Monitoring based on sampling state-variables has historically been disregarded as a runtime monitoring approach, for good reason: without the assumption of synchrony between the monitor and observed software, monitoring via sampling may lead to false positives and false negatives [11]. For example, consider the property $(0; 1; 1)^*$, written as a regular expression, denoting the sequence of values a monitored variable may take. If the monitor samples the variable at the inappropriate time, then both false negatives (the monitor erroneously rejects the sequence of values) and false positives (the monitor erroneously accepts the sequence) are possible. For example, if the actual sequence of values is $0, 1, 1, 0, 1, 1$, then an observation of $0, 1, 1, 1, 1$ is a false negative by skipping a value, and if the actual sequence is $0, 1, 0, 1, 1$, then an observation of $0, 1, 1, 0, 1, 1$ is a false positive by sampling a value twice.

However, in a hard real-time context, sampling is a suitable strategy. Under the assumption that the monitor and the observed program share a global clock and a static periodic schedule, while false positives are possible, false negatives are not. A false positive is possible, for example, if the program does not execute according to its schedule but just happens to have the expected values when sampled. If a monitor samples an unacceptable sequence of values, then either the program is in error, the monitor is in error, or they are not synchronized, all of which are faults to be reported.

Most of the popular runtime monitoring frameworks inline monitors in the observed program to avoid the aforementioned problems with sampling. However, inlining monitors changes the real-time behavior of the observed program, perhaps in unpredicable ways. Recalling our four criteria from Section 3.1, monitors that introduce such unpredictability are not a viable solution for ultra-critical hard real-time systems. In a sampling-based approach, the monitor can be integrated as a separate scheduled process during available time slices (this is made possible by generating efficient constant-time monitors). Indeed, sampling-based monitors may even be scheduled on a separate processor (albeit doing so requires additional synchronization mechanisms), ensuring time and space partitioning from the observed programs. Such an architecture may even be necessary if the monitored program is physically distributed.

Recent work in RV investigates the use of time-triggered sampling for arbitrary programs [13,5,31], which is a harder problem than in our hard real-time context.

4 Copilot Language

Copilot is embedded into the functional programming language Haskell [16], and a working knowledge of Haskell is necessary to use Copilot effectively. Copilot is a pure declarative language; i.e., expressions are free of side-effects and satisfies referential transparency. A program written in Copilot, which from now on will be referred to as a *specification*, has a cyclic behavior, where each cycle consists of a fixed series of steps:

- Sample external variables, arrays, and functions.
- Update internal variables.
- Fire external triggers. (In case the specification is violated.)

We refer to a single cycle as an *iteration*.

All transformation of data in Copilot is propagated through streams. A stream is an infinite, ordered sequence of values which must conform to the same type. E.g., we have the stream of Fibonacci numbers:

$$s_{fib} = \{0, 1, 1, 2, 3, 5, 8, 13, 21, \dots\}$$

We denote the nth value of the stream s as $s(n)$, and the first value in a sequence s as $s(0)$. For example, for s_{fib} we have that $s_{fib}(0) = 0$, $s_{fib}(1) = 1$, $s_{fib}(2) = 1$, and so forth.

Constants as well as arithmetic, boolean, and relational operators are lifted to work pointwise on streams:

```
x :: Stream Int32
x = 5 + 5

y :: Stream Int32
y = x * x

z :: Stream Bool
z = x == 10 && y < 200
```

Here the streams x, y, and z are simply *constant streams*:
$$x \rightsquigarrow \{10, 10, 10, \dots\}, \; y \rightsquigarrow \{100, 100, 100, \dots\},$$
$$z \rightsquigarrow \{T, \; T, \; T, \; \dots\}$$

Two types of *temporal* operators are provided, one for delaying streams and one for looking into the future of streams:

```
(++) :: [a] -> Stream a -> Stream a
drop :: Int -> Stream a -> Stream a
```

where the type signature for the function is given in curried form after the double colon. Here `xs ++ s` prepends the list `xs` at the front of the stream `s`. For example the stream `w` defined as follows, given our previous definition of `x`:

```
w = [5,6,7] ++ x
```

evaluates to the sequence $w \rightsquigarrow \{5, 6, 7, 10, 10, 10, \dots\}$. The expression `drop k s` skips the first `k` values of the stream `s`, returning the remainder of the stream. For example we can skip the first two values of `w`:

```
u = drop 2 w
```

which yields the sequence $u \rightsquigarrow \{7, 10, 10, 10, \dots\}$.

4.1 Streams as Lazy-Lists

A key design choice in Copilot is that streams should mimic *lazy lists*. In Haskell, the lazy-list of natural numbers can be programmed like this:

```
nats_ll :: [Int32]
nats_ll = [0] ++ zipWith (+) (repeat 1) nats_ll
```

where `zipWith` is a higher-order function that takes a function and two lists as parameters and produces a new list formed by applying the given function to the elements of the two input lists at the same position in each list. As both constants and arithmetic operators are lifted to work pointwise on streams in Copilot, there is no need for `zipWith` and `repeat` when specifying the stream of natural numbers:

```
nats :: Stream Int32
nats = [0] ++ (1 + nats)
```

In the same manner, the lazy-list of Fibonacci numbers can be specified as follows:

```
fib_ll :: [Int32]
fib_ll = [1, 1] ++ zipWith (+) fib_ll (drop 1 fib_ll)
```

In Copilot we simply throw away `zipWith`:

```
fib :: Stream Int32
fib = [1, 1] ++ (fib + drop 1 fib)
```

Copilot specifications must be *causal*, informally meaning that stream values cannot depend on future values. For example, the following stream definition is allowed:

```
f :: Stream Word64
f = [0,1,2] ++ f

g :: Stream Word64
g = drop 2 f
```

But if instead `g` is defined as `g = drop 4 f`, then the definition is disallowed. While an analogous stream is definable in a lazy language, we bar it in Copilot, since it requires future values of `f` to be generated before producing values for `g`. This is not possible since Copilot programs may take inputs in real-time from the environment (see Section 4.5).

4.2 Functions on Streams

Given that constants and operators work pointwise on streams, we can use Haskell as a macro-language for defining functions on streams. The idea of using Haskell as a macro language is powerful since Haskell is a general-purpose higher-order functional language.

Example 1 We define the function, `even`, which given a stream of integers returns a boolean stream which is true whenever the input stream contains an even number, as follows:

```
even :: Stream Int32 -> Stream Bool
even x = x 'mod' 2 == 0
```

Applying `even` on `nats` (defined above) yields the sequence $\{T, F, T, F, T, F, \dots\}$.

If a function is required to return multiple results, we simply use plain Haskell tuples:

Example 2 We define complex multiplication as follows:

```
mul_comp
  :: (Stream Double, Stream Double)
  -> (Stream Double, Stream Double)
  -> (Stream Double, Stream Double)
(a, b) 'mul_comp' (c, d) = (a * c - b * d, a * d + b * c)
```

Here `a` and `b` represent the real and imaginary part of the left operand, and `c` and `d` represent the real and imaginary part of the right operand.

4.3 Stateful Functions

In addition to pure functions, such as `even` and `mul_comp`, Copilot also facilitates *stateful* functions. A *stateful* function is a function which has an internal state, e.g. as a latch (as in electronic circuits) or a low/high-pass filter (as in a DSP).

Example 3 We consider a simple latch, as described in [12], with a single input and a boolean state. Whenever

x_i:	y_{i-1}:	y_i:
F	F	F
F	T	T
T	F	T
T	T	F

```
latch :: Stream Bool -> Stream Bool
latch x = y
  where
  y = if x then not z else z
  z = [False] ++ y
```

Fig. 1: A latch. The specification is provided at the left and the implementation is provided at the right.

the input is true the internal state is reversed. The operational behavior and the implementation of the latch is shown in Figure 1.[1]

Example 4 We consider a resettable counter with two inputs, `inc` and `reset`. The input `inc` increments the counter and the input `reset` resets the counter. The internal state of the counter, `cnt`, represents the value of the counter and is initially set to zero. At each cycle, i, the value of cnt_i is determined as shown in the left table in Figure 2.

4.4 Types

Copilot is a typed language, where types are enforced by the Haskell type system to ensure generated C programs are well-typed. Copilot is *strongly typed* (i.e., type-incorrect function application is not possible) and *statically typed* (i.e., type-checking is done at compile-time). The base types are Booleans, unsigned and signed words of width 8, 16, 32, and 64, floats, and doubles. All elements of a stream must belong to the same base type. These types have instances for the class `Typed a`, used to constrain Copilot programs. The constraining type classes are listed to the left of the `=>` in a function definition.

We provide a `cast` operator

```
cast :: (Typed a, Typed b) => Stream a -> Stream b
```

that casts from one type to another. The cast operator is only defined for casts that do not lose information, so an unsigned word type `a` can only be cast to another unsigned type at least as large as `a` or to a signed word type strictly larger than `a`. Signed types cannot be cast to unsigned types but can be cast to signed types at least as large.

4.5 Interacting With the Target Program

All interaction with the outside world is done by sampling *external symbols* and by evoking *triggers*. External

symbols are symbols that are defined outside Copilot and which reflect the visible state of the target program that we are monitoring. They include variables, arrays, and functions (with a non-void return type). Analogously, triggers are functions that are defined outside Copilot and which are evoked when Copilot needs to report that the target program has violated a specification constraint.

Sampling. A Copilot specification is *open* if defined with external symbols in the sense that values must be provided externally at runtime. To simplify writing Copilot specifications that can be interpreted and tested, constructs for external symbols take an optional environment for interpretation.

External variables are defined by using the `extern` construct:

```
extern :: Typed a => String -> Maybe [a] -> Stream a
```

where a value of type `Maybe a` is either empty (`Nothing`) or a value of type `a` (`Just a`). The function `extern` takes the name of an external variable, a possible Haskell list to serve as the environment for the interpreter from which the monitor will draw the values that would otherwise be sampled from the external program, and generates a stream by sampling the variable at each clock cycle. For example,

```
sumExterns :: Stream Word64
sumExterns = let ex1 = extern "e1" (Just [0..])
                 ex2 = extern "e2" Nothing
             in  ex1 + ex2
```

is a stream that takes two external variables `e1` and `e2` and adds them. The first external variable contains the infinite list `[0,1,2,...]` of values for use when interpreting a Copilot specification containing the stream. The other variable contains no environment (`sumExterns` must have an environment for both of its variables to be interpreted).

Sometimes, type inference cannot infer the type of an external variable. For example, in the stream definition

```
extEven :: Stream Bool
extEven = e0 'mod' 2 == 0
  where e0 = externW8 "x" Nothing
```

---

[1] In order to use conditionals (if-then-else's) in Copilot specifications, as in Figures 1 and 2, the GHC language extension `RebindableSyntax` must be set on.

inc_i:	$reset_i$:	cnt_i:
F	F	cnt_{i-1}
$*$	T	0
T	F	$cnt_{i-1} + 1$

```
counter :: Stream Bool -> Stream Bool
            -> Stream Int32
counter inc reset = cnt
  where
  cnt = if reset then 0
          else if inc then z + 1
                  else z
  z = [0] ++ cnt
```

Fig. 2: A resettable counter. The specification is provided at the left and the implementation is provided at the right.

the type of `extern "x"` is ambiguous, since it cannot be inferred from a Boolean stream and we have not given an explicit type signature. For convenience, typed `extern` functions are provided, e.g., `externW8` or `externI64` denoting an external unsigned 8-bit word or signed 64-bit word, respectively. In general it is best practice to define external symbols with top-level definitions, e.g.,

```
e0 :: Stream Word8
e0 = extern  "e0" (Just [2,4..])
```

so that the symbol name and its environment can be shared between streams.

Besides variables, external arrays and arbitrary functions can be sampled. The external array construct has the type

```
externArray :: (Typed a, Typed b, Integral a)
            => String -> Stream a -> Int
            -> Maybe [[a]] -> Stream b
```

The construct takes (1) the name of an array, (2) a stream that generates indexes for the array (of integral type), (3) the fixed size of the array, and (4) possibly a list of lists that is the environment for the external array, representing the sequence of array values. For example,

```
extArr :: Stream Word32
extArr = externArray "arr1" arrIdx size
          (Just $ repeat (permutations [0,1,2]))
  where
  arrIdx :: Stream Word8
  arrIdx = [0] ++ (arrIdx + 1) `mod` size

  size = 3
```

`extArr` is a stream of values drawn from an external array containing 32-bit unsigned words. The array is indexed by an 8-bit variable. The index is ensured to be less than three by using modulo arithmetic. The environment provided produces an infinite list of all the permutations of the list `[0,1,2]`.[2]

Example 5 Say we have defined a lookup-table (in C99) of a discretized continuous function that we want to use within Copilot:

---

[2] The function `permutations` comes from the Haskell standard library `Data.list`.

```
double someTable[42] = { 3.5, 3.7, 4.5, ... };
```

We can use the table in a Copilot specification as follows:

```
lookupSomeTable :: Stream Word16 -> Stream Double
lookupSomeTable idx =
  externArray "someTable" idx 42 Nothing
```

Given the following values for `idx`, $\{1, 0, 2, 2, 1, \dots\}$, the output of `lookupSomeTable idx` would be

$$\{3.7, 3.5, 4.5, 4.5, 3.7, \dots\}$$

Finally, the constructor `externFun` takes (1) a function name, (2) a list of arguments, and (3) a possible list of values to provide its environment.

```
externFun :: Typed a => String -> [FunArg]
          -> Maybe [a] -> Stream a
```

Each argument to an external function is given by a Copilot stream. For example,

```
func :: Stream Word16
func = externFun "f" [arg e0, arg nats] Nothing
  where
  e0 = externW8 "x" Nothing
  nats :: Stream Word8
  nats = [0] ++ nats + 1
```

samples a function in C that has the prototype

```
uint16_t f(uint8_t x, uint8_t nats);
```

Both external arrays and functions must, like external variables, be defined in the target program that is monitored. Additionally, external functions must be without side effects, so that the monitor does not cause undesired side-effects when sampling functions. Finally, to ensure Copilot sampling is not order-dependent, external functions cannot contain streams containing other external functions or external arrays in their arguments, and external arrays cannot contain streams containing external functions or external arrays in their indexes. They can both take external variables, however.

Triggers. Triggers, the only mechanism for Copilot streams to effect the outside world, are defined by using the `trigger` construct:

```
trigger :: String -> Stream Bool -> [TriggerArg] -> Spec
```

The first parameter is the name of the external function, the second parameter is the guard which determines when the trigger should be evoked, and the third parameter is a list of arguments which is passed to the trigger when evoked. Triggers can be combined into a specification by using the *do*-notation:

```
spec :: Spec
spec = do
  trigger "f" (even nats) [arg fib, arg (nats * nats)]
  trigger "g" (fib > 10) []
  let x = externW32 "x" Nothing
  trigger "h" (x < 10) [arg x]
```

The order in which the triggers are defined is irrelevant.

Example 6 We consider an engine controller with the following property: If the temperature rises more than 2.3 degrees within 0.2 seconds, then the fuel injector should not be running. Assuming that the global sample rate is 0.1 seconds, we can define a monitor that surveys the above property:

```
propTempRiseShutOff :: Spec
propTempRiseShutOff =
  trigger "over_temp_rise"
    (overTempRise && running) []

  where
  max = 500 -- maximum engine temperature

  temps :: Stream Float
  temps = [max, max, max] ++ temp

  temp = extern "temp" Nothing

  overTempRise :: Stream Bool
  overTempRise = drop 2 temps > (2.3 + temps)

  running :: Stream Bool
  running = extern "running" Nothing
```

Here, we assume that the external variable `temp` denotes the temperature of the engine and the external variable `running` indicates whether the fuel injector is running. The external function `over_temp_rise` is called without any arguments if the temperature rises more than 2.3 degrees within 0.2 seconds and the engine is not shut off. Notice there is a latency of one tick between when the property is violated and when the guard becomes true.

4.6 Explicit Sharing

Copilot facilitates sharing in expressions by the *local*-construct:

```
local
  :: (Typed a, Typed b)
  => Stream a
  -> (Stream a -> Stream b)
  -> Stream b
```

```
s1 = let x = nats + nats
     in x * x
s2 = local (nats + nats) $
     \ x -> x * x
```

Fig. 3: Implicit sharing (s1) versus explicit sharing (s2).

The local construct works similar to *let*-bindings in ordinary Haskell. A stream defined using a local construct typically has the structure

$$s = s' \ \$ \ \backslash x \ \text{->} \ e$$

where `s'` is a stream, $ is Haskell's application operator that forces function application to associate to the right, and the `e` is a lambda expression. From a semantic point of view, the streams `s1` and `s2` from Figure 3 are identical. As we will see in Section 4.7, however, certain advanced Copilot programs may force the compiler to build syntax trees that blow up exponentially. In such cases, using explicit sharing helps to avoid this problem.

4.7 Extended Example: The Boyer-Moore Majority-Vote Algorithm

In this section we demonstrate how to use Haskell as an advanced macro language on top of Copilot by implementing an algorithm for solving the voting problem in Copilot.

Reliability in mission critical software is often improved by replicating the same computations on separate hardware and by doing a vote in the end based on the output of each system. The majority vote problem consists of determining if in a given list of votes there is a candidate that has more than half of the votes, and if so, of finding this candidate.

The Boyer-Moore Majority Vote Algorithm [24,15] solves the problem in linear time and constant memory. It does so in two passes: The first pass chooses a candidate; and the second pass asserts that the found candidate indeed holds a majority.

Without going into details of the algorithm, the first pass can be implemented in Haskell as shown in Figure 4. The second pass, which simply checks that a candidate has more than half of the votes, is straightforward to implement and is shown in Figure 5. E.g. applying `majorityPure` on the string `AAACCBBCCCBCC` yields `C`, which `aMajorityPure` can confirm is in fact a majority.

When implementing the majority vote algorithm for Copilot, we can use reuse almost all of the code from the Haskell implementation. However, as functions in

```
majorityPure :: Eq a => [a] -> a
majorityPure []     = error "majorityPure: empty list!"
majorityPure (x:xs) = majorityPure' xs x 1

majorityPure' []     can _   = can
majorityPure' (x:xs) can cnt =
  let
    can' = if cnt == 0 then x else can
    cnt' = if cnt == 0 || x == can then succ cnt else pred cnt
  in
    majorityPure' xs can' cnt'
```

Fig. 4: The first pass of the majority vote algorithm in Haskell.

```
aMajorityPure :: Eq a => [a] -> a -> Bool
aMajorityPure xs can = aMajorityPure' 0 xs can > length xs `div` 2

aMajorityPure' cnt []     _   = cnt
aMajorityPure' cnt (x:xs) can =
  let
    cnt' = if x == can then cnt+1 else cnt
  in
    aMajorityPure' cnt' xs can
```

Fig. 5: The second pass of the majority vote algorithm in Haskell.

```
majority :: (Eq a, Typed a) => [Stream a] -> Stream a
majority []     = error "majority: empty list!"
majority (x:xs) = majority' xs x 1

majority' []     can _   = can
majority' (x:xs) can cnt =
  local
    (if cnt == 0 then x else can) $
      \ can' ->
        local (if cnt == 0 || x == can then cnt+1 else cnt-1) $
          \ cnt' ->
            majority' xs can' cnt'
```

Fig. 6: The first pass of the majority vote algorithm in Copilot.

Copilot are macros that are expanded at compile time, care must be taken in order to avoid an explosion in the code size. Hence, instead of using Haskell's built-in *let*-blocks, we use explicit sharing, as described in Section 4.6. The Copilot implementations of the first and the second pass are given in Figure 6 and Figure 7 respectively. Comparing the Haskell implementation with the Copilot implementation, we see that the code is almost identical, except for the type signatures and the explicit sharing annotations.

5 Tools

As a framework for runtime verification, Copilot comes with a variety of tools to support the generation of runtime monitor code as well as to the assist in their assurance. The Copilot toolchain is depicted in Figure 8. A Copilot program is transformed into the Copliot Core language via process called reification. An interpreter providing an executable semantics for the Core language is provided as an integral part of the toolchain. The interpreter can be used to prototype Copilot programs as well as for verification and validation purposes. Reified programs can be printed using the custom pretty-printing tool. Currently, two backends, Atom and SBV, are included in the toolchain. Both tools translate Copilot core language programs into a Haskell-hosted EDSL for C code generation. In addition, a custom Quickcheck engine and a test harness are provided along with support of model checkers to aide in verifying the generated C monitors. We will demonstrate these tools and their usage in the remainder of this section. A more thorough discussion about the verification approach can be found in [27].

```
aMajority :: (Eq a, Typed a) => [Stream a] -> Stream a -> Stream Bool
aMajority xs can = aMajority' 0 xs can > (fromIntegral (length xs) `div` 2)

aMajority' cnt []      _    = cnt
aMajority' cnt (x:xs) can =
  local
    (if x == can then cnt+1 else cnt) $
      \ cnt' ->
        aMajority' cnt' xs can
```

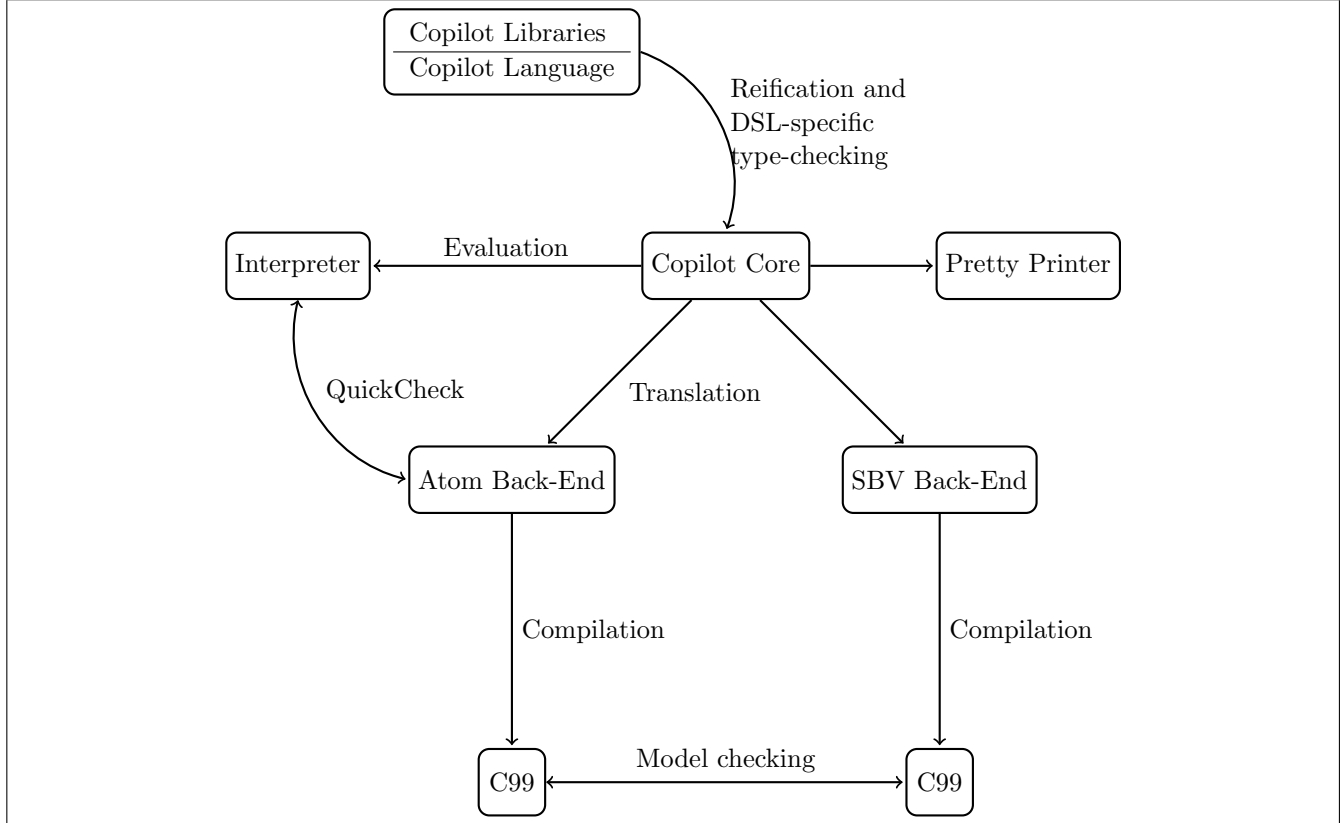Fig. 7: The second pass of the majority vote algorithm in Copilot.

Fig. 8: The Copilot toolchain.

5.1 Pretty-Printing

Pretty-printing is straightforward. For some specification spec,

```
prettyPrint spec
```

returns the specification after static macro expansion. Pretty-printing can provide some indication about the complexity of the specification to be evaluated. Specifications that are built by recursive Haskell programs (e.g., the majority voting example in Section 4.7) can generate expressions that are large. Large expressions can take significant time to interpret or compile.

5.2 Interpreting Copilot

The copilot interpreter is invoked as follows (e.g. within GHCI, the GHC compiler's interpreter for Haskell):

```
GHCI> interpret 10 propTempRiseShutOff
```

The first argument to the function *interpret* is the number of iterations that we want to evaluate. The third argument is the specification (of type Spec) that we wish to interpret.

The interpreter outputs the values of the arguments passed to the trigger, if its guard is true, and `--` otherwise. For example, consider the following Copilot program:

```
spec = do
  trigger "trigger1" (even nats) [arg nats, arg $ odd nats]
  trigger "trigger2" (odd nats) [arg nats]
```

where `nats` is the stream of natural numbers, and `even` and `odd` are functions that take a stream and return whether the point-wise values are even or odd, respectively. The output of

```
interpret 10 spec
```

is as follows:

```
trigger:   trigger2:
(0,false)  --
--         (1)
(2,false)  --
--         (3)
(4,false)  --
--         (5)
(6,false)  --
--         (7)
(8,false)  --
--         (9)
```

Sometimes it is convenient to observe the behavior of a stream without defining a trigger. We can do so declaring an *observer*. For example:

```
spec :: Spec
spec = observer ''obs'' nats
```

can be interpreted using

```
interpret 5 spec
```

as usual. Observers can be combined in larger Copilot programs. For example, consider the following:

```
spec :: Spec
spec = do
  let x = externW8 "x" (Just [0..])
  trigger "trigger" true [arg $ x < 3]
  observer "debug_x" x
```

Interpreting `spec` as follows

```
interpret 10 spec
```

yields

```
trigger:   debug_x:
(true)     0
(true)     1
(true)     2
(false)    3
(false)    4
(false)    5
(false)    6
(false)    7
(false)    8
(false)    9
```

5.3 Compiling Copilot

Compiling the engine controller from Example 6 is straight-forward. First, we pick a back-end to compile to. Currently, two back-ends are implemented, both of which generate constant-time and constant-space C code. One back-end is called *copilot-c99* and targets the Atom language,[3] originally developed by Tom Hawkins at Eaton Corp. for developing control systems. The second back-end is called *copilot-sbv* and targets the SBV language[4], originally developed by Levent Erkök. SBV is primarily used as an interface to SMT solvers [3] and also contains a C-code generator. Both languages are open-source.

The two back-ends are installed with Copilot, and they can be imported, respectively, as

```
import Copilot.Compile.C99
```

and

```
import Copilot.Compile.SBV
```

After importing a back-end, the interface for compiling is as follows:[5]

```
reify spec >>= compile defaultParams
```

(The compile function takes a parameter to rename the generated C files; `defaultParams` is the default, in which there is no renaming.)

The compiler now generates two files:

- "copilot.c" —
- "copilot.h" —

The file named "copilot.h" contains prototypes for all external variables, functions, and arrays, and contains a prototype for the "step"-functions which evaluates a single iteration.

```
/* Generated by Copilot Core v. 0.1 */

#include <stdint.h>
#include <stdbool.h>

/* Triggers (must be defined by user): */

void over_temp_rise();

/* External variables (must be defined by user): */

extern float temp;
extern bool running;

/* Step function: */

void step();
```

Using the prototypes in "copilot.h" we can build a driver as follows:

```
/* driver.c */
#include <stdio.h>
#include "copilot.h"
```

---

[3] http://hackage.haskell.org/package/atom
[4] http://hackage.haskell.org/package/sbv
[5] Two explanations are in order: (1) `reify` allows sharing in the expressions to be compiled [14], and `>>=` is a higher-order operator that takes the result of reification and "feeds" it to the compile function.

```
bool running = true;
float temp = 1.1;

void over_temp_rise()
{
  printf("The trigger has been evoked!\n");
}

int main (int argc, char const *argv[])
{
  int i;

  for (i = 0; i < 10; i++)
  {
    printf("iteration: %d\n", i);
    temp = temp * 1.3;
    step();
  }

  return 0;
}
```

Running "gcc copilot.c driver.c -o prop" gives a program "prop", which when executed yields the following output:

```
iteration: 0
iteration: 1
iteration: 2
iteration: 3
iteration: 4
iteration: 5
iteration: 6
iteration: 7
The trigger has been evoked!
iteration: 8
The trigger has been evoked!
iteration: 9
The trigger has been evoked!
```

5.4 QuickCheck

QuickCheck [9] is a library originally developed for Haskell such that given a property, it generates random inputs to test the property. We provide a similar tool for checking Copilot specifications. Currently, the tool is implemented to check the copilot-c99 back-end against the interpreter. The tool generates a random Copilot specification, and for some user-defined number of iterations, the output of the interpreter is compared against the output of the compiled C program. The user can specify weights to influence the probability at which expressions are generated.

The copilot QuickCheck tool is installed with Copilot and assuming the binary is in the path, it is executed as

```
copilot-c99-qc
```

5.5 Verification

"Who watches the watchmen?" Nobody. For this reason, monitors in ultra-critical systems are the last line of defense and cannot fail. Here, we outline our approach to generate high-assurance monitors. First, as mentioned, the compiler is statically and strongly typed, and by implementing an eDSL, much of the infrastructure of a well-tested Haskell implementation is reused. We have described our custom QuickCheck engine. We have tested millions of randomly-generated programs between the compiler and interpreter with this approach.

Additionally, Copilot includes a tool to generate a driver to prove the equivalence between the copilot-c99 and copilot-sbv back-ends that each generate C code (similar drivers are planned for future back-ends). To use the driver, first import the following module:

```
import qualified Copilot.Tools.CBMC as C
```

(We import it using the `qualified` keyword to ensure no name space collisions.) Then in GHCI, just like with compilation, we execute

```
reify spec >>= C.genCBMC C.defaultParams
```

This generates two sets of C sources, one compiled through the copilot-c99 back-end and one through the copilot-sbv back-end. In addition, a driver (that is, a `main` function) is generated that executes the code from each back-end. The driver has the following form:

```
int main (int argc, char const *argv[])
{
  int i;

  for (i = 0; i < 10; i++)
  {
    sampleExterns();
    atm_step();
    sbv_step();
    assert(atm_i == sbv_i);
  }

  return 0;
}
```

This driver executes the two generated programs for ten iterations, which is the default value. That default can be changed; for example:

```
reify spec >>=
  C.genCBMC C.defaultParams {C.numIterations = 20}
```

The above executes the generated programs for 20 executions.

The verification depends on an open-source model-checker for C source-code originally developed at Carnegie Mellon University [10]. A license for the tool is available. [6] CBMC must be downloaded and installed separately; CBMC is actively maintained at the time of

---

[6] http://www.cprover.org/cbmc/LICENSE. It is the user's responsibility to ensure their use conforms to the license.

writing, and is available for Windows, Linux, and Mac OS.

CBMC symbolically executes a program. With different options, CBMC can be used to check for arithmetic overflow, buffer overflow/underflow, floating-point NaN results, and division by zero. Additionally, CBMC can attempt to verify arbitrary `assert()` statements placed in the code. In our case, we wish to verify that on each iteration, for the same input variables, the two back-ends have the same state.

CBMC proves that for all possible inputs, the two programs have the same outputs for the number of iterations specified. The time-complexity of CBMC is exponential with respect to the number of iterations. Furthermore, CBMC cannot guarantee equivalence beyond the fixed number of iterations.

After generating the two sets of C source files, CBMC can be executed on the file containing the driver; for example,

```
cbmc cbmc_driver.c
```

6 Case Studies: Monitoring Avionics

We describe two case studies in which we have used Copilot monitors.

6.1 Pitot Tube Fault-Tolerance

In commercial aircraft, airspeed is commonly determined using pitot tubes that measure air pressure. The difference between total and static air pressure is used to calculate airspeed. Pitot tube subsystems have been implicated in numerous commercial aircraft incidents and accidents, including the 2009 Air France crash of an A330 [2], motivating our case study.

We have developed a platform resembling a real-time air speed measuring system with replicated processing nodes, pitot tubes, and pressure sensors to test distributed Copilot monitors with the objective of detecting and tolerating software and hardware faults, both of which are purposefully injected. The platform and its inclusion in an Edge 540 test aircraft, is depicted in Figure 9.

The high-level procedure of our experiment is as follows: (1) we sense and sample air pressure from the aircraft's pitot tubes; (2) apply a conversion and calibration function to accommodate different sensor and analog-to-digital converter (ADC) characteristics; (3) sample the C variables that contain the pressure values on a hard real-time basis by Copilot-generated monitors; and (4) execute Byzantine fault-tolerant voting and fault-tolerant averaging on the sensor values to detect arbitrary hardware component failures and keep consistent values among good nodes.

We sample five pitot tubes, attached to the wings of an Edge 540 subscale aircraft. The pitot tubes provide total and static pressure that feed into one MPXV5004DP and four MPXV7002DP differential pressure sensors (Figure 10). The processing nodes are four STM 32 microcontrollers featuring ARM Cortex M3 cores which are clocked at 72 Mhz (the number of processors was selected with the intention of creating applications that can tolerate one Byzantine processing node fault [20]). The MPXV5004DP serves as a shared sensor that is read by each of the four processing nodes; each of the four MPXV7002DP pressure sensors is a local sensor that is only read by one processing node.

Monitors communicate over dedicated point-to-point bidirectional serial connections. With one bidirectional serial connection between each pair of nodes, the monitor bus and the processing nodes form a complete graph. All monitors on the nodes run in synchronous steps; the clock distribution is ensured by a master hardware clock. (The clock is a single point of failure in our prototype hardware implementation; a fully fault-tolerant system would execute a clock-synchronization algorithm.)

Each node samples its two sensors (the shared and a local one) at a rate of 16Hz. The microcontroller's timer interrupt that updates the global time also periodically calls a Copilot-generated monitor which samples the ADC C-variables of the monitored program, conducts Byzantine agreements, and performs fault-tolerant votes on the values. After a complete round of sampling, agreements, and averaging, an arbitrary node collects and logs intermediate values of the process to an SD-card.

We tested the monitors in five flights. In each flight we simulated one node having a permanent Byzantine fault by having one monitor send out pseudo-random differing values to the other monitors instead of the real sampled pressure. We varied the number of injected benign faults by physically blocking the dynamic pressure ports on the pitot tubes. In addition, there were two "control flights", leaving all tubes unmodified.

The executed sampling, agreement, and averaging is described as follows:

1. Each node samples sensor data from both the shared and local sensors.
2. Each monitor samples the C variables that contain the pressure values and broadcasts the values to every other monitor, then relays each received value to monitors the value did not originate from.
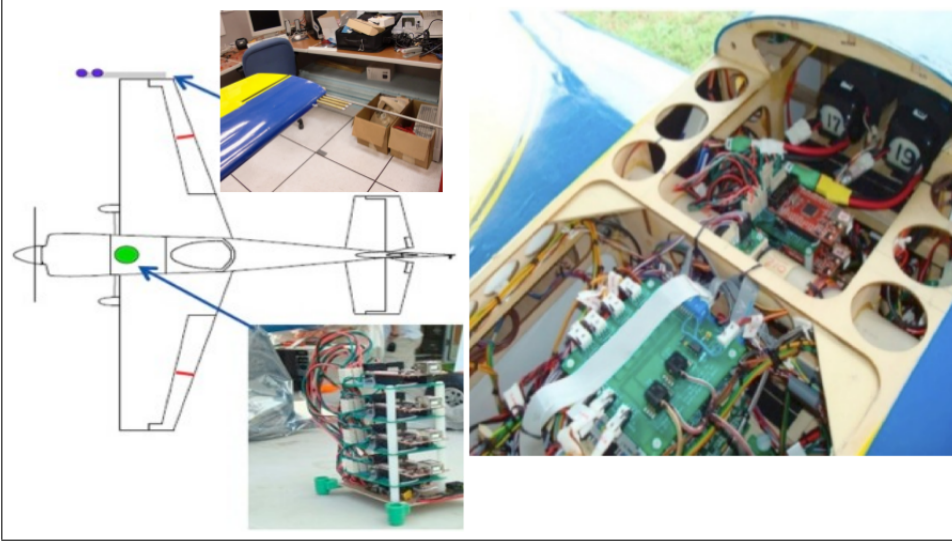
Fig. 9: Stack configuration in the Edge 540 aircraft.
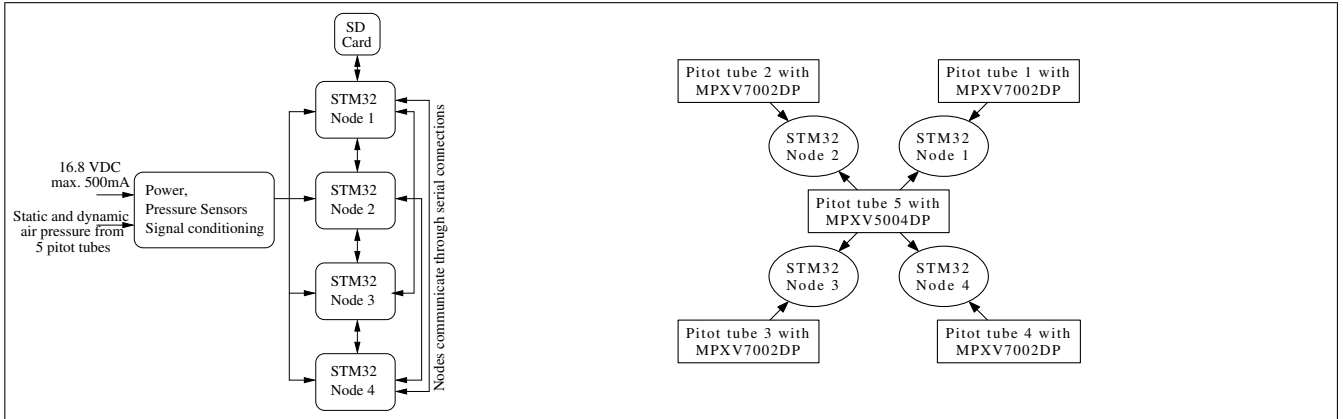

Fig. 10: Hardware stack and pitot tube configuration.

3. Each monitor performs a majority vote (as described in Section 4.7) over the three values it has for every other monitor of the shared sensor (call this $maj_i(S)$ for node i) and the local sensor (call this $maj_i(L)$ for node i).

4. Copilot-generated monitors then compute a *fault-tolerant average*. In our implementation, we remove the least and greatest elements from a set, and average the remaining elements. For each node i and nodes $j \neq i$, fault-tolerant averages are taken over four-element sets: (1) $ftAvg(S) = \{S_i\} \cup \{maj_j(S)\}$ where S_i is i's value for the shared sensor.

5. Another fault-tolerant average is taken over a five-element set, where the two least and two greatest elements are removed (thus returning the median value). The set contains the fault-tolerant average over the shared sensor described in the previous step ($ftAvg(S)$), the node's local sensor value L_i, and

$\{maj_j(L)\}$, for $j \neq i$. Call this final fault-tolerant average $ftAvg$.

6. Finally, time-stamps, sensor values, majorities and their existences are collected by one node and recorded to an SD card for off-line analysis.

The graphs in Figure 11 depict four scenarios in which different faults are injected. In each scenario, there is a software-injected Byzantine faulty node present. What varies between the scenarios are the number of physical faults. In Figure 11(a), no physical faults are introduced; in Figure 11(b), one benign fault has been injected by putting a cap over the total pressure probe of one local tube.[7] In Figure 11(c), in addition to the capped tube, sticky tape is placed over another tube, and in Figure 11(d), sticky tape is placed over two tubes in addition to the capped tube.

---

[7] Tape left on the static pitot tube of Aeroperú Flight 603 in 1996 resulted in the death of 70 passengers and crew [19].
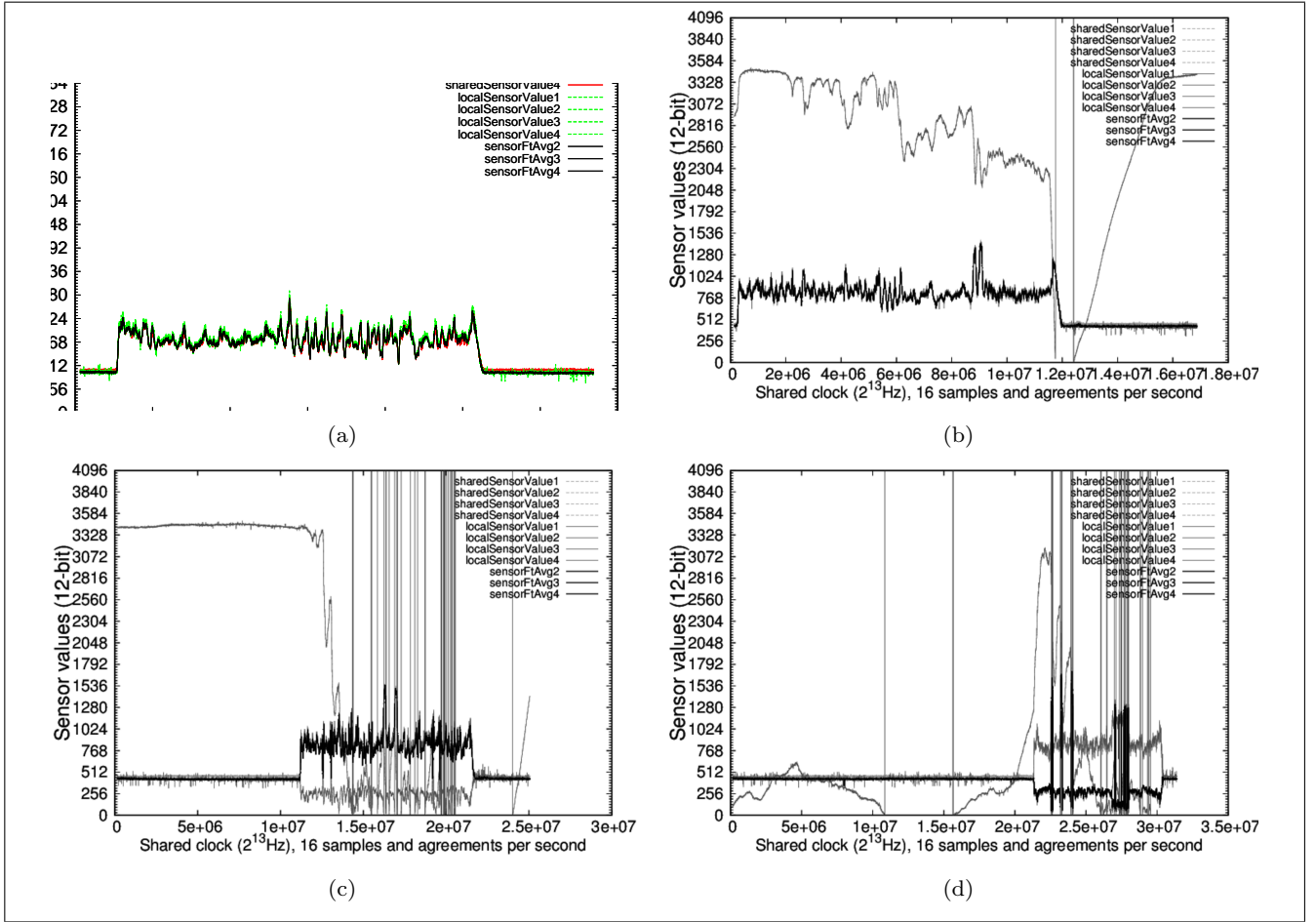
Fig. 11: Logged pressure sensor, voted and averaged data.

The graphs depict the air pressure difference data logged at each node and the voted and averaged outcome of the 3 non-faulty processing nodes. The gray traces show the recorded sensor data $S_1, \ldots, S_4$, and the calibrated data of the local sensors $L_1, \ldots, L_4$. The black traces show the final agreed and voted values $ftAvg$ of the three good nodes.

In every figure except for Figure 11(d), the black graphs approximate each other, since the fault-tolerant voting allows the nodes to mask the faults. This is despite wild faults; for example, in Figure 11(b), the cap on the capped tube creates a positive offset on the dynamic pressure as well as turbulences and low pressure on the static probes. At 1.2E7 clock ticks, the conversion and calibration function of the stuck tube results in an underflowing value. In Figure 11(d), with only two non-faulty tubes out of five left, $ftAvg$ is not able to choose a non-faulty value reliably anymore. All nodes still agree on a consistent—but wrong—value.

6.2 MAVLink Monitoring

The MAVLink (Micro Air Vehicle Link[8]) protocol consists of a set of messages to be sent between small air vehicles and ground stations. Althought it can be used to send messages on parameters like wind speed or attitude, the usual applications of MAVLink are in avionic systems with an autopilot. MAVLink is used by groundstation software packages, like QGroundControl, Happy Killmore Ground Control Station, the Ardupilot Mega Planner and autopilot systems like PIXHAWK or the Ardupilot Mega. MAVLink commands and messages of version 2 of the protocol are specified in XML files that contain common and groundstation/autopilot specific packet definitions.

We have implemented portions of the common set MAVLink protocol as Copilot monitors and have executed them on binary MAVLink log files. Additionally, we have executed the monitor in real-time on three flights of an Edge R540T subscale aircraft to analyze

---

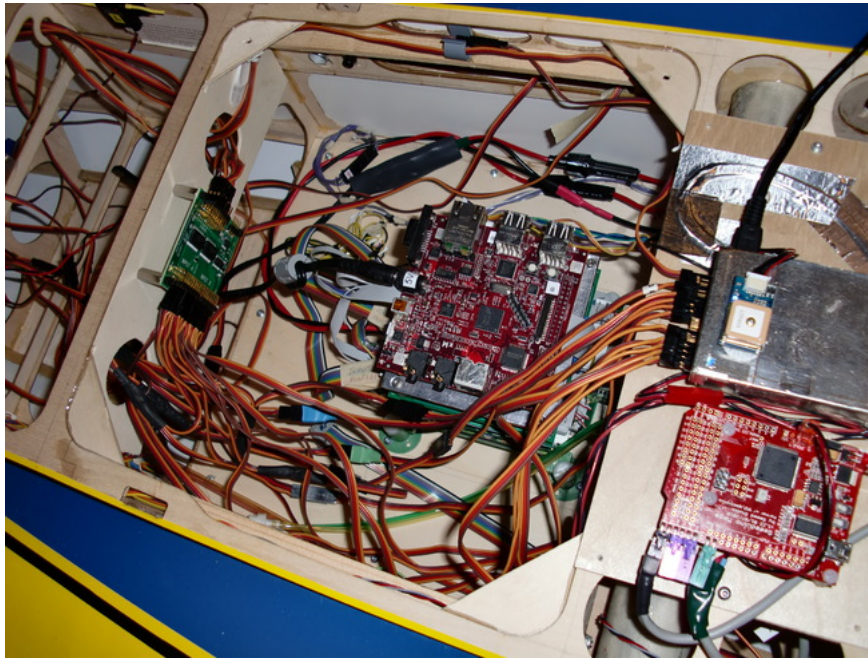[8] http://qgroundcontrol.org/mavlink/start

Fig. 12: Beagle Board executing the MAVLink monitor.

MAVLink packets from an Ardupilot Mega. The configuration in the Edge is depicted in Figure 12. In the center is a Beagleboard xM that executes the monitors described below. On the right-hand side, inside the silver box, is an Arduino Mega board that runs the Ardupilot autopilot. The red board below the silver box is a Seeeduino, that is used as a serial hub that connects the XBee radio to the groundstation, the Beagleboard and the Ardupilot.

The layout of a packet frame in MAVLink version is listed in Table 1. The example column lists a packet of the MAVLink heartbeat type (message `id 0x00` and payload length three) as it was captured from a Zig-Bee link between an ArduPilot Mega and an ArduPilot Mega Planner groundstation. Heartbeat messages are sent in regular intervals and are used to keep track of different vehicles as they appear and leave the visibility of receiving nodes. The three payload bytes stand for the type of aircraft (`0x01` - fixed wing), the type of the autopilot (`0x03` - Ardupilot) and the MAVLink version (`0x02`).

According to Table 1, we define some protocol specific sizes and limits, next to their constant Copilot stream versions:

```
startSequenceSize  = 1
startSequenceSize' = constant startSequenceSize

headerSize         = 6
headerSize'        = constant headerSize

crcSize            = 2
```

```
crcSize'           = constant crcSize

maxPayloadLength   = 255
maxPayloadLength'  = constant maxPayloadLength

maxPacketLength    = headerSize  + maxPayloadLength
                                  + crcSize
maxPacketLength'   = headerSize' + maxPayloadLength'
                                  + crcSize'
```

To analyze incoming packets, we define an input stream that has a long enough initial array to keep one MAVLink packet of maximum length.[9] In each tick, the next MAVLink byte is sampled from the C variable `extern_input` and shifted into the array from the right:

```
-- The input stream, allows dropping up to
--  the maximum packet length
inputStream :: Stream Word32
inputStream = replicate maxPacketLength 0 ++ externInput

-- The actual MAVLink input
externInput :: Stream Word32
externInput = extern "extern_input" Nothing
```

Further, we define where in a packet to access header fields and payload, according to Table 1:

```
payloadLength         = drop 1 inputStream
packetLength          = headerSize' + payloadLength
                                    + crcSize'
packetSequenceNumber  = drop 2 inputStream
systemID              = drop 3 inputStream
componentID           = drop 4 inputStream
```

[9] At the time of this writing, Copilot did not handle streams of arrays. Modeling the protocol as a stream of `Word32`s, as we explain herein, is inefficient, resulting in a large specification.

Table 1: MAVLink packet fields.

Byte Index	Content	Value	Example
0	Packet start sign	$0x55$, ASCII: U	0x55
1	Payload length n	$0 - 255$	0x03
2	Packet sequence	$0 - 255$	0x13
3	System ID	$1 - 255$	0x01
4	Component ID	$0 - 255$	0x01
5	Message ID	$0 - 255$	0x00
6 to $n + 6$	Payload data	$0 - 255$ per byte	0x01, 0x03, 0x02
$n + 7$ to $n + 8$	Checksum over bytes 1..(n+6)	$0 - 65535$	0x32, 0xb7

```
messageID            = drop 5 inputStream
payload              = drop 6 inputStream
```

The MAVLink checksum is a modification of the checksum used in the X.25 protocol; it uses the same calculation as the X.25 cyclic redundancy check, but does not invert the final remainder. A Copilot function that takes an initial remainder `r` and 8 bits of the input stream `d`, then calculates a new remainder by dividing `d` by the X.25 polynomial $x^{16} + x^{12} + x^5 + 1$, is listed below:

```
mavlinkCrcUpdate :: Stream Word32 -> Stream Word32
                                  -> Stream Word32
mavlinkCrcUpdate r d =
let d'   = d    .&. 0xff
    tmp  = d' .^. ( r .&. 0xff )
    tmp' = tmp .^. ( shiftL 4 tmp .&. 0xff )
in foldl1 (.^.) [ shiftR 8 r,    shiftL 8 tmp'
              , shiftL 3 tmp', shiftR 4 tmp' ]
```

where `.&.` and `.^` denote the bitwise and a exclusive or operators respectively.

Left-folding the `mavlinkCrcUpdate` function with an initial value `crcInit = 0xffff` into the initial array, starting from the second packet byte up to the maximum packet length (and keeping the intermediate CRC results), is achieved by the Copilot `nscanl` library function.[10]

```
crcStreams :: [ Stream Word32 ]
crcStreams = nscanl
        ( maxPacketLength - startSequenceSize )
        mavlinkCrcUpdate crcInit
        ( drop 1 inputStream )
```

The `crcStreams` list contains the CRC values of all prefixes of a possible packet. The CRC over all values of a valid packet, excluding the start sign and including a valid CRC at the end of the packet, will be zero:

```
crcIndex :: Stream Word32
crcIndex = headerSize' + payloadLength
```

---

[10] Copilot's `nscanl` is a fixed-length (of n) analogue of the Haskell `scanl` function in Haskell, such that `scanl f z [x1, x2, ...] == [z, z ‘f‘ x1, (z ‘f‘ x1) ‘f‘ x2, ...]`.

```
        - startSequenceSize' + crcSize'

crc :: Stream Word32
crc = crcStreams !! crcIndex

crcValid :: Stream Bool
crcValid = crc == 0
```

Given the definitions to check the CRC values of a packet, we can now check for valid packets:

```
startMatch  = inputStream == 0x55
validPacket = startMatch && crcValid
```

The communication between an autopilot and a ground-station runs over a ZigBee link. In case of dropped radio packets, there is no guarantee a receiver will not (on reconnection of the radio link) interpret a wrong packet, i.e., the start sign `0x55` may appear anywhere in a packet and a following length/CRC pair can form a valid "ghost packet" (i.e., a packet that is contained within an actually sent packet or that spans multiple actually sent packets).

We define a stream called `analyzingPacket`, that signals a running analysis of a valid packet as:

```
-- The analyzingPacket function signals if we recognized
-- a valid packet and have not reached the end yet
analyzingPacket = analyzingPacket' > 1
  where analyzingPacket' = [ 0 ] ++ mux
            ( validPacket && not analyzingPacket )
            -- set the counter
            ( ( drop 1 inputStream
              + headerSize'
              + crcSize' )
            -- count down the packet length
            ( mux ( analyzingPacket' == 0 )
                0
                ( analyzingPacket' - 1 ) ) )
```

We then can recognize ghost packets by checking for valid packets that appear while we are in the process of analyzing a packet, provided that `analyzingPacket` starts out on an actually sent packet and not on a ghost packet:

```
ghostPacket = analyzingPacket && validPacket
```

We ran the `ghostPacket` monitor on about 660 megabytes of binary MAVLink logs recorded during months of hardware-in-the-loop testing of an Ardupilot Mega in an Edge 540T subscale model. The ghostPacket monitor fired a trigger 32 times.

On a lost radio connection that sets in after the dropout, the receiver has a chance to misinterpret such a ghost packet. For a receiver not to accept a ghost packet, it can relate the sequence numbers of packets to its actual system time. If such measures are not implemented, an autopilot may receive commands over MAVLink that might lead to unexpected behaviors.

MAVLink carries a number of sensor values. We wrote a simple monitor that analyses the payload of `GLOBAL_POSITION_INT` messages to retrieve a trajectory of flight:

```
packet mId = validPacket && messageID == mId

packetWithLength mId pLen = packet mId
                         && payloadLength == pLen

-- the global position in integer values has
-- message id 73  and payload length 18
globalPositionINT = packetWithLength 73 18
```

The first 12 bytes of the payload of a `GLOBAL_POSITION_INT` messages are interpreted as three `Word32` values of latitude, longitude and altitude [11]. Reconstruction of the position is done by 3 streams, globalPositionIntLat, globalPositionIntLon and globalPositionIntAlt:

```
s3 = ( .<<. ( constant 24 :: Stream Word32 ) )
s2 = ( .<<. ( constant 16 :: Stream Word32 ) )
s1 = ( .<<. ( constant 8  :: Stream Word32 ) )

globalPositionIntLat :: Stream Word32
globalPositionIntLat = let l1 = drop 0 payload
                           l2 = drop 1 payload
                           l3 = drop 2 payload
                           l4 = drop 3 payload
                       in s3 l1 + s2 l2 + s1 l3 + l4

globalPositionIntLon :: Stream Word32
globalPositionIntLon = let l1 = drop 4 payload
                           l2 = drop 5 payload
                           l3 = drop 6 payload
                           l4 = drop 7 payload
                       in s3 l1 + s2 l2 + s1 l3 + l4

globalPositionIntAlt :: Stream Word32
globalPositionIntAlt = let l1 = drop 8  payload
                           l2 = drop 9  payload
                           l3 = drop 10 payload
                           l4 = drop 11 payload
                       in s3 l1 + s2 l2 + s1 l3 + l4
```

where `.<<.` denotes the left shift operator.

The streams become parameters of a globalPositionInt trigger:

```
trigger "globalPositionINT" globalPositionIN
             [ arg globalPositionIntLat
             , arg globalPositionIntLon
             , arg globalPositionIntAlt ]
```

The globalPositionINT trigger C function logs each set of three values. We ran the monitors on three flights and plotted the trajectories.

Consider the two graphs shown in Figure 13 and 14, respectively, which graph the latitude, longitude, and altitude of the aircraft during two flights. Comparing the graphs, in Figure 14, the graph has small discrete "steps" resulting from the quantization error that is caused by the GPS receiver losing tracking, updating positions at a lower rate. (The disturbance was caused by an unknown condition, but we were nonetheless able to monitor its effect.) The MAVLink `GLOBAL_POSITION_INT` packet type we analyzed contains latitude and longitude as given by the GPS and altitude as a combination of barometric altitude and GPS altitude. Because the latitude and longitude are not updated at the usual rate, the most recently-seen values together with the changed altitude (the altitude changes because–while GPS altitude is not updated–barometric altitude is) and causes the stair effect.

6.3 Discussion

The purpose of the case studies is to test the feasibility of using Copilot-generated monitors in a realistic setting to "bolt on" fault-tolerance to a system that otherwise lacks that capability.

To give a sense of code-sizes, in the pitot tube monitoring case-study, the Copilot agreement monitor is around 200 lines, and the generated real-time C code is nearly 4,000 lines. In the MAVLink case-study, the Copilot monitor is around 300 lines, with an additional 350 lines of support C code, implementing triggers and the CRC.[12] The Copilot monitor generates about 2500 lines of real-time C code.

7 Conclusions and Remaining Challenges

Ultra-critical systems need RV. Our primary goals in this paper are to (1) motivate this need, (2) describe one approach for RV in the ultra-critical domain, (3) and present evidence for its feasibility.

The approach we have described in this report is not without shortcomings, which present opportunities for future research.

---

[11] Latitude and longitude in degrees, altitude in meters.

[12] When streams of arrays are implemented in Copilot, the CRC can be derived from a Copilot specification.
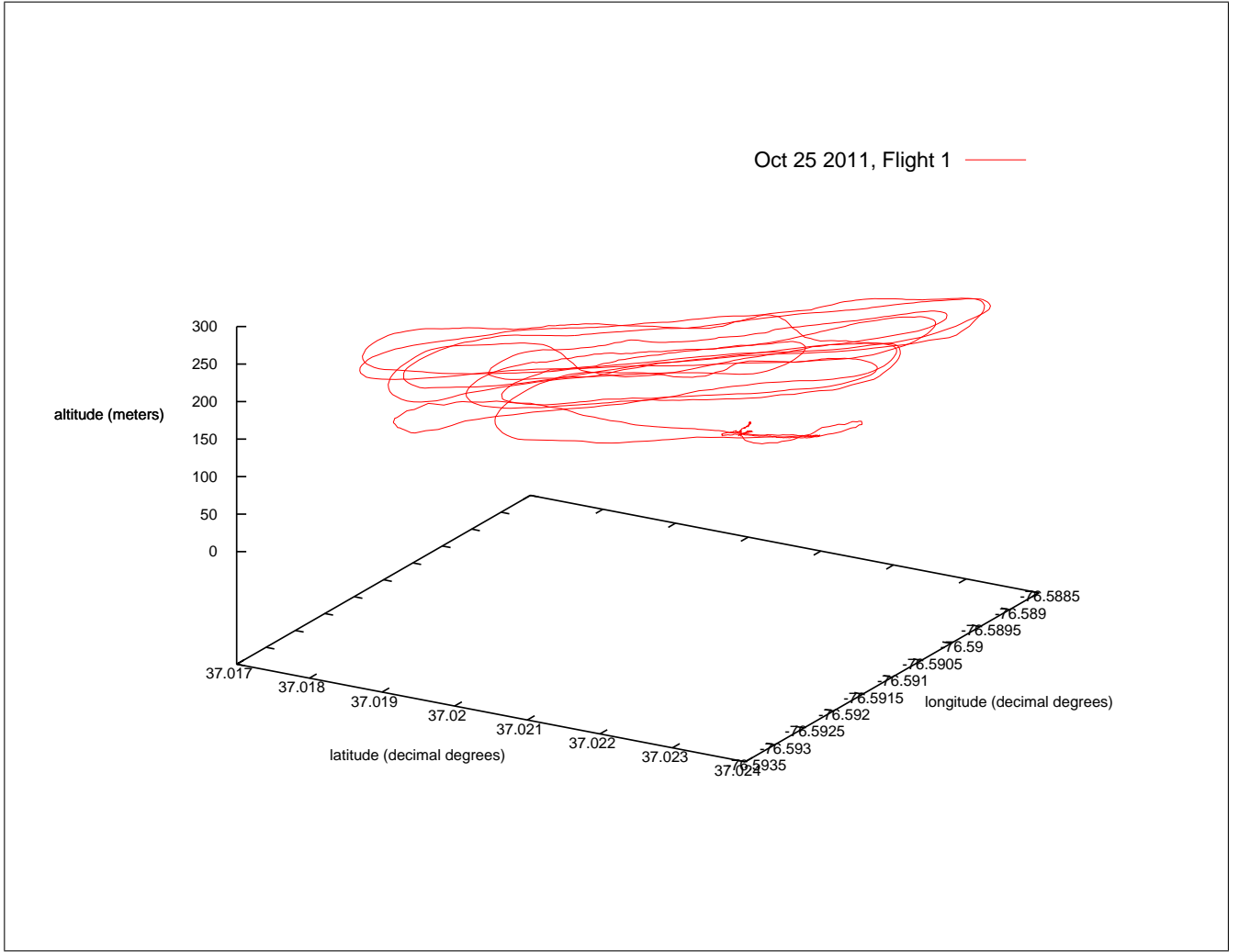
Fig. 13: Flight 1.

eDSL efficiency. First, we have demonstrated that the embedded DSL approach is powerful, turning regular programming on its head: while Copilot is simple, its macro language is a higher-order functional language! One disadvantage of this approach is that with a powerful macro language, it is easy to build up large expressions—much larger than would be built in a conventional programming language. For example, the Boyer-Moore voting algorithm described in Section 4.7 is compiled into a single Copilot expression. The use of explicit sharing (Section 4.6) reduces the cost of computation by ensuring sub-expressions are not needlessly recomputed, but if the sub-expressions themselves are expensive to compute, the entire expression becomes expensive.

Techniques to improve the efficiency of evaluating eDSLs are needed. Fortunately, monitoring code is terse, in general.

Assurance The lightweight approach to monitor assurance discussed in Section 5 is described in more detail in [27]. The current framework for front-end/back-end testing is built on Quickcheck, which does not provide coverage testing capability. Given the criticality of the monitors, DO-178 [28] would require that MC/DC testing be performed if they were to be employed in industrial avionics. Adding this capability to the toolchain is a challenge for future work.

A more heavyweight approach to verification is exemplified by the CompCert [21] C compiler. Adapting a similar approach, we could generate proofs or certificates that the C monitors created by the back-end meet the specification given by the Copilot operational semantics and then use the CompCert compiler to provide a certificate that the executable created during compilation is correct.
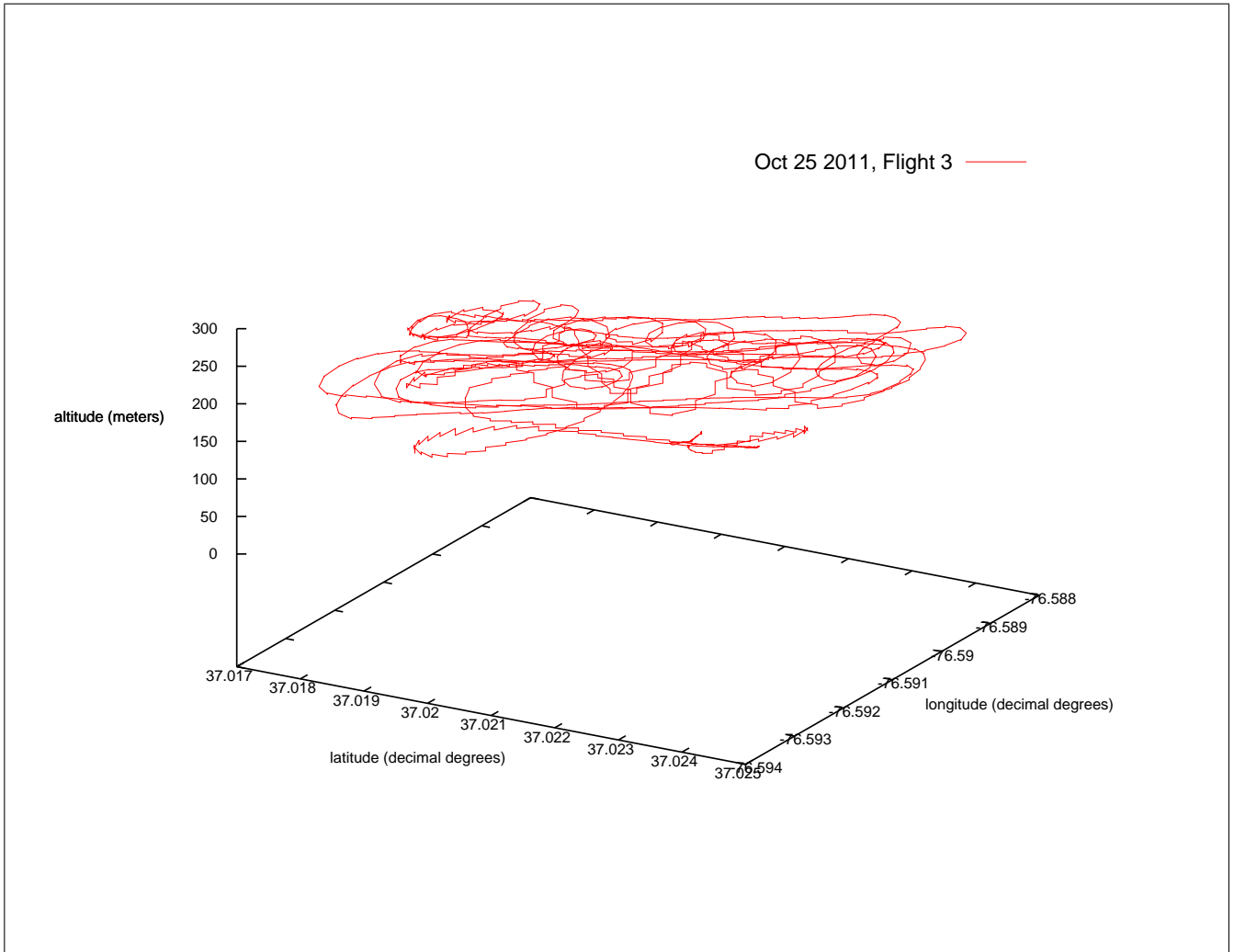
Fig. 14: Flight 2.

Scheduling monitors. In the experiments described in Section 6, we use hardware interrupts to ensure monitors run at fixed intervals. This technique works in practice and obviates the need for an underlying operating system to handle scheduling. However, we must ensure that monitors execute quickly (so that the monitored system does not miss other interrupts), and we need to ensure that the monitor has been given sufficient time to execute. With the current set of code generators, worst-case execution time is easy to compute, as there is just one control-path through the code (that is, worst-case execution time is equal to nominal execution time).

Safety-Critical hard real-time systems typically employ real-time operating systems (RTOS) to manage the schedule. Copilot has been ported to an ARINC 653 compliant RTOS and experiments are being planned to test applications being monitored by Copilot programs, where they are both scheduled by the OS using algorithms such as rate-monotonic scheduling.

The only model of time in Copilot monitors, like other synchronous languages, is the tick. The tick is an abstract model of time that gets mapped to a real-time duration by the underlying hardware. The duration of a tick matters when specifying monitors: the property

The value of x must satisfy -0.5 <= x - x' <= 0.5, where x' is the value of x exactly one second ago.

requires building a stream of values. If a tick is one second long, then the specification

```
prop = (x - x') <= 5 && (x - x') <= (-5)
  where
  x  = [0] ++ e0
  x' = drop 1 x
  e0 = externI32 "x" Nothing
```

If a tick is a half-second, we must use `drop 2 ...`, and so on. Thus, monitors may be hardware/scheduler de-

pendent. It would help the specifier to lift the abstraction level, so she can write properties in terms of real-time.

Other language features. In analyzing protocol streams, reconstructing values out of the payload of a packet from incoming bytes is necessary. Copilot currently lacks casting operations to do this. Adding a general set of casting functions that includes different byte orders, bit orders and number representations would help on monitoring protocols.

Steering. We have not addressed the *steering* problem of how to address faults once they are detected. Steering is critical at the application level; for example, if an RV monitor detects that a control system has violated its permissible operational envelope.

Faults. We have built a system to detect both hardware and software (logical) faults. Stochastic methods might be used to distinguish random hardware faults from systematic faults, as the steering strategy for responding to each differs [30].

Conclusions. Research developments in RV have potential to improve the reliability of ultra-critical systems. Research into runtime monitoring for hard real-time distributed systems has been under-represented in the community, but we hope a growing number of RV researchers address this application domain.

References

1. FAA system handbook (2000). Available at http://www.faa.gov/library/manuals/aviation/risk_management/ss_handbook/

2. Aviation Today: More pitot tube incidents revealed. Aviation Today (2011). Available at http://www.aviationtoday.com/regions/usa/More-Pitot-Tube-Incidents-Revealed_72414.html

3. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability Modulo Theories, chap. 26, pp. 825–885. Frontiers in Artificial Intelligence and Applications. IOS Press (2009)

4. Bergin, C.: Faulty MDM removed. NASA Spaceflight.com (2008). Available at http://www.nasaspaceflight.com/2008/05/sts-124-frr-debate-outstanding-issues-faulty-mdm-removed/. (Downloaded Nov 28, 2008)

5. Bonakdarpour, B., Navabpour, S., Fischmeister, S.: Sampling-based runtime verification. In: 17th Intl. Symposium on Formal Methods (FM) (2011)

6. Bureau, A.T.S.: In-flight upset event 240 Km North-West of Perth, WA Boeing Company 777-200, 9M-MRG 1 August 2005. ATSB Transport Safety Investigation Report (2007). Aviation Occurrace Report - 200503722

7. Butler, R.W., Finelli, G.B.: The infeasibility of quantifying the reliability of life-critical real-time software. IEEE Transactions on Software Engineering **19**, 3–12 (1993)

8. Chen, F., Roşu, G.: Java-MOP: a monitoring oriented programming environment for Java. In: 11th Intl. Conf. on Tools and Algorithms for the construction and analysis of systems (TACAS'05), *LNCS*, vol. 3440, pp. 546–550. Springer (2005)

9. Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of haskell programs. In: ACM SIGPLAN Notices, pp. 268–279. ACM (2000)

10. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, pp. 168–176. Springer (2004)

11. Dwyer, M., Diep, M., Elbaum, S.: Reducing the cost of path property monitoring through sampling. In: Proceedings of the 23rd International Conference on Automated Software Engineering, pp. 228–237 (2008)

12. Farhat, H.: Digital design and computer organization. No. v. 1 in Digital Design and Computer Organization. CRC Press (2004)

13. Fischmeister, S., Ba, Y.: Sampling-based program execution monitoring. In: ACM International conference on Languages, compilers, and tools for embedded systems (LCTES), pp. 133–142 (2010)

14. Gill, A.: Type-safe observable sharing in Haskell. In: Proceedings of the 2009 ACM SIGPLAN Haskell Symposium (2009)

15. Hesselink, W.H.: The Boyer-Moore majority vote algorithm (2005)

16. Jones, S.P. (ed.): Haskell 98 Language and Libraries: The Revised Report. http://haskell.org/ (2002)

17. Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I., Sokolsky, O.: Formally specified monitoring of temporal properties. In: 11th Euromicro Conference on Real-Time Systems, pp. 114–122 (1999)

18. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. Communications of the ACM (CACM) **53**(6), 107–115 (2010)

19. Ladkin, P.B.: News and comment on the Aeroperu b757 accident; AeroPeru Flight 603, 2 october 1996 (2002). Online article RVS-RR-96-16. Available at http://www.rvs.uni-bielefeld.de/publications/Reports/aeroperu-news.html

20. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems **4**, 382–401 (1982)

21. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**, 107–115 (2009)

22. Leveson, N.G., Turner, C.S.: An investigation of the Therac-25 accidents. Computer **26**, 18–41 (1993)

23. Macaulay, K.: ATSB preliminary factual report, in-flight upset, Qantas Airbus A330, 154 Km West of Learmonth, WA, 7 October 2008. Australian Transport Safety Bureau Media Release (2008). Available at `http://www.atsb.gov.au/newsroom/2008/release/2008_45.aspx`

24. Moore, S.J., Boyer, R.S.: MJRTY - A Fast Majority Vote Algorithm. Tech. Rep. 1981-32, Institute for Computing Science, University of Texas (1981)

25. Nuseibeh, B.: Soapbox: Ariane 5: Who dunnit? IEEE Software **14**(3), 15–16 (1997)

26. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: A hard real-time runtime monitor. In: Runtime Verification (RV), vol. 6418, pp. 345–359. Springer (2010)

27. Pike, L., Wegmann, N., Niller, S., Goodloe, A.: Experience report: Do-it-yourself high-assurance compiler. In: Proceedings of the 17th ACM SIGPLAN Conference on Functional Programming. ACM (2012)

28. RTCA: Software considerations in airborne systems and equipment certification. RTCA, Inc. (1992). RCTA/DO-178B

29. Rushby, J.: Software verification and system assurance. In: Intl. Conf. on Software Engineering and Formal Methods (SEFM), pp. 3–10. IEEE (2009)

30. Sammapun, U., Lee, I., Sokolsky, O.: RT-MaC: runtime monitoring and checking of quantitative and probabilistic properties. In: Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 147–153 (2005)

31. Stoller, S.D., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S.A., Zadok, E.: Runtime verification with state estimation. In: Proc. of the 2nd International Conference on Runtime Verification (RV'11). San Fransisco, CA (2011). (**Won best paper award**)