Innovations Syst Softw Eng (2013) 9:235–255
DOI 10.1007/s11334-013-0223-x

SI: SWHM

Copilot: monitoring embedded systems

Lee Pike · Nis Wegmann · Sebastian Niller · Alwyn Goodloe

Received: 5 January 2012 / Accepted: 15 July 2013 / Published online: 24 August 2013

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.

Keywords Runtime verification · Monitoring · Real-time systems · Distributed systems

L. Pike (✉)
Galois, Inc., Portland, USA
e-mail: leepike@gmail.com; leepike@galois.com

N. Wegmann
University of Copenhagen, Copenhagen, Denmark
e-mail: wegmann@diku.dk

S. Niller
Evertz Microsystems, Burlington, Canada
e-mail: sniller@evertz.com

A. Goodloe
NASA, Hampton, USA
e-mail: a.goodloe@nasa.gov

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 [39]. The justification for the requirement 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 10,000 lines of code represents the state-of-the-art, taking 20 engineer-years to achieve [26]. Nearly 20 years ago, Butler and Finelli [11] showed that testing alone cannot verify the reliability of ultra-critical software.

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 also be a 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 Sect. 2, we motivate the need for runtime verification for ultra-critical systems. In Sect. 3, we describe constraints and approaches to RV for embedded systems. In Sect. 4, we present the language of Copilot, our approach for ultra-critical RV. We describe the tools (e.g., the interpreter and compiler back-ends) associated with Copilot in Sect. 5. We present two case-studies applying Copilot to avionic systems in Sect. 6. Related work is given in Sect. 7, and we make final remarks in Sect. 8.

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 [30] and the Ariane 5 Flight 501 disaster [34]. 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.

2.1 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 [7]. 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 [29]), which was not tolerated and forced an emergency launch abortion.

2.2 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 [10]. 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.

2.3 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 m in 20 s (a subsequent less severe pitch was also made) [31]. The accident required the hospitalization of 14 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":

- *Functionality* the RV system cannot change the target's behavior (unless the target has violated a specification).
- *Certifiability* the RV system must not make re-certification (e.g., DO-178B [37]) of the target onerous.
- *Timing* the RV system must not interfere with the target's timing.
- *SWaP* 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) [25] and Monitor Oriented Programming (MOP) [13] frameworks). While runtime verification is not currently considered in the context of certification, there are reasons for doing so, discussed by Rushby [38]. Not modifying the observed program could simplify the introduction of RV into certified systems, reducing the need to re-evaluate code that is being monitored. Source code instrumentation can modify both the control flow of the instrumented program as well as its timing properties.

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 [35].

The final constraint, SWaP, applies both to memory (embedded processors may have just a few kilobytes of available memory) and 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 [16]. 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. Often, the purpose of real-time programs is to deliver output signals at a predicable rate and properties of interest are generally data-flow oriented. In this context, and 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 Sect. 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 and control-flow oriented properties [18,9,41], which is a harder problem than in our hard real-time context.

4 Copilot language

Copilot is embedded into the functional programming language Haskell [24], 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 satisfy 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:

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

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 $s_{\text{fib}}(0) = 0$, $s_{\text{fib}}(1) = 1$, $s_{\text{fib}}(2) = 1$, and so forth.

Copilot declarations provide an optional type and a definition for an identifier. For example,

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

gives the identifier x the type Stream Int32 and defines x to be 5. All Copilot expressions have the type Stream t, where t is some base-type for the stream (we discuss Copilot types in Sect. 4.4).

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* such that

x yields $\{10, 10, 10, \ldots\}$
y yields $\{100, 100, 100, \ldots\}$
z yields $\{\mathrm{T},\ \mathrm{T},\ \mathrm{T},\ \ldots\}$

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
```

Here xs ++ s prepends the list xs at the front of the stream s. For example, the stream w is defined as follows, given our previous definition of x:

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

evaluates to the sequence $\{5, 6, 7, 10, 10, 10, \ldots\}$. 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 $\{7, 10, 10, 10, \ldots\}$.

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. The Haskell function repeat creates an infinite list with each element having the value of the first parameter. 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 Sect. 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 $\{T, F, T, F, T, F, \ldots\}$.

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

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*

$\mathtt{inc}_i$:	$\mathtt{reset}_i$:	$\mathtt{cnt}_i$:
F	F	$\mathtt{cnt}_{i-1}$
$*$	T	0
T	F	$\mathtt{cnt}_{i-1}+1$

```
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*

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 [17], with a single input and a boolean state. Whenever the input is true, the internal state is reversed. The operational behavior and the implementation of the latch are shown in Fig. 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 $\mathtt{cnt}_i$ is determined as shown in the left table in Fig. 2.

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

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 interactions with the outside world are 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.

4.5.1 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 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 and a possible (Haskell) list of values to serve as the "environment" for the interpreter, since in a compiled monitor, these variable values come from the sampled program. The interpreter generates a stream by sampling the variable at each clock cycle, modeled as the elements in the list. 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). If `ex1` and `ex2` both were defined with the lists `[0,1,2,...]`, then the output of interpreting `sumExterns` would be `[0,2,4,...]`.

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 = extern ``x'' Nothing
```

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 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:

```
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`, $\{1, 0, 2, 2, 1, \ldots\}$, the output of `lookupSomeTable idx` would be

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

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,

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

```
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.

4.5.2 Triggers

Triggers, the only mechanism for Copilot streams to effect the outside world, are defined 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 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 ◦ within 0.2 s, then the fuel injector should not be running. Assuming that the global sample rate is 0.1 s, we can define a monitor that surveys the above property:

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

  where
```

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

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

```
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 ◦ within 0.2 s and the engine has not been shut off. Note that 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
```

The local construct works similar to *let*-bindings in Haskell. For example, from a semantic point of view, the streams `s1` and `s2` in Fig. 3 are equivalent. However, by sharing expression definitions, the size of the Copilot program can be dramatically reduced. Printing the expressions results in something like the following:

```
s1 = (nats + nats) * (nats + nats)
s2 = let x = nats + nats in
     x * x
```

Note that in `s1`, because the sharing is at the *Haskell* level, the expressions are inlined, resulting in replication in the resulting expression.

```
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

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 [33,23] 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 Fig. 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 Fig. 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 Copilot are macros that are expanded at compile time, care must be taken to avoid an explosion in the expression size; thus, note the use of the `local` construct. The Copilot implementations of the

```
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

first and the second pass are given in Figs. 6 and 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 assist in assurance of the monitors. The Copilot toolchain is depicted in Fig. 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 back-ends are included in the toolchain, which we describe in Sect. 5.3.Both back-ends translate Copilot Core-language programs into a Haskell-hosted embedded domain-specific language (eDSL) for C code generation. In addition, a custom Quickcheck engine and a test harness are provided along with support for model checkers to aid 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 [36].

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 indications about the complexity of the specification to be evaluated. Specifications that are built by recursive Haskell programs (e.g., the majority voting example in Sect. 4.7) can generate expressions that are large.

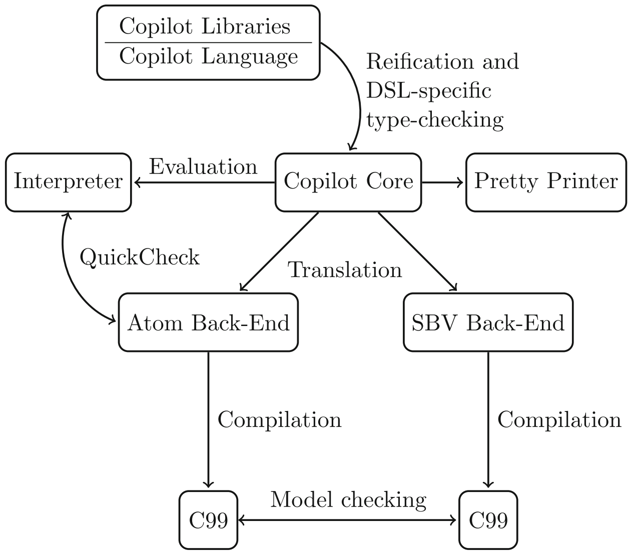

Fig. 8 The Copilot toolchain

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:

```
trigger1:   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 straightforward. 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 [6] 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): */
```

[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 [19], and >>= is a higher-order operator that takes the result of reification and “feeds” it to the compile function.

```
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''

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:
    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 [14] 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 [15]. A license for the tool is available.[6] CBMC must be downloaded and installed separately; CBMC is actively maintained at the time of 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 [4], motivating our case study.

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

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 Fig. 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 (Fig. 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 [29]). 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 test hardware implementation; a fully fault-tolerant system would execute a clock-synchronization algorithm.)

Each node samples its two sensors (the shared sensor 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 out 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

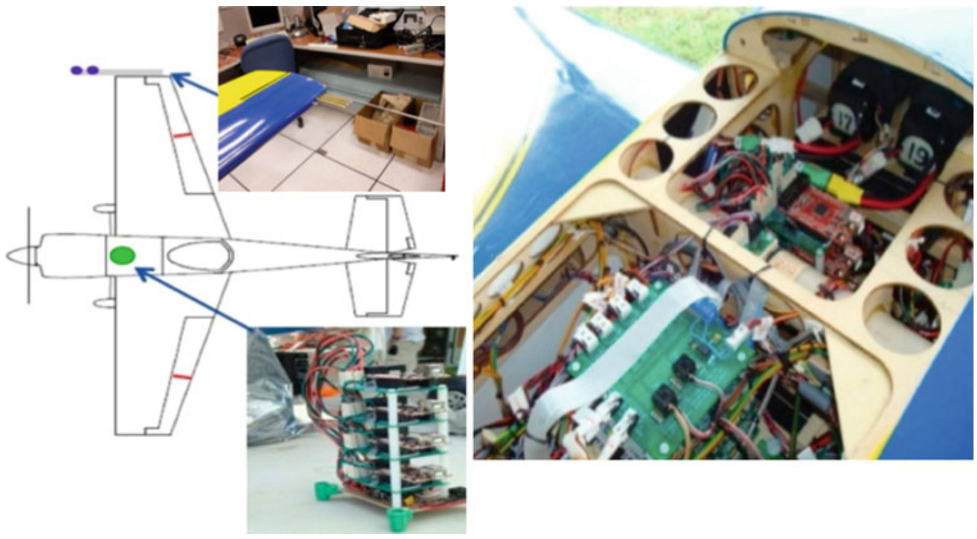

Fig. 9 Stack configuration in the Edge 540 aircraft

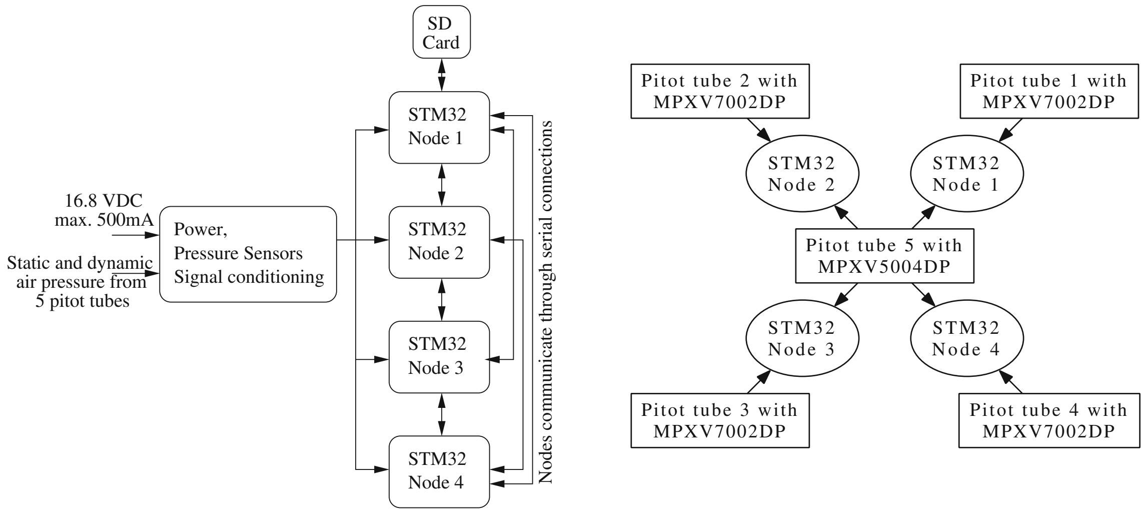
SD Card
STM32 Node 1
STM32 Node 2
STM32 Node 3
STM32 Node 4
16.8 VDC max. 500mA
Static and dynamic air pressure from 5 pitot tubes
Power, Pressure Sensors Signal conditioning
Nodes communicate through serial connections
Pitot tube 2 with MPXV7002DP
Pitot tube 1 with MPXV7002DP
Pitot tube 5 with MPXV5004DP
Pitot tube 3 with MPXV7002DP
Pitot tube 4 with MPXV7002DP

Fig. 10 Hardware stack and pitot tube configuration

addition, there were two “control flights”, leaving all tubes unmodified.

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

1. Each node samples sensor data from both the shared and local sensor.
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 that the value did not originate from.
3. Each monitor performs a majority vote (as described in Sect. 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 Fig. 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

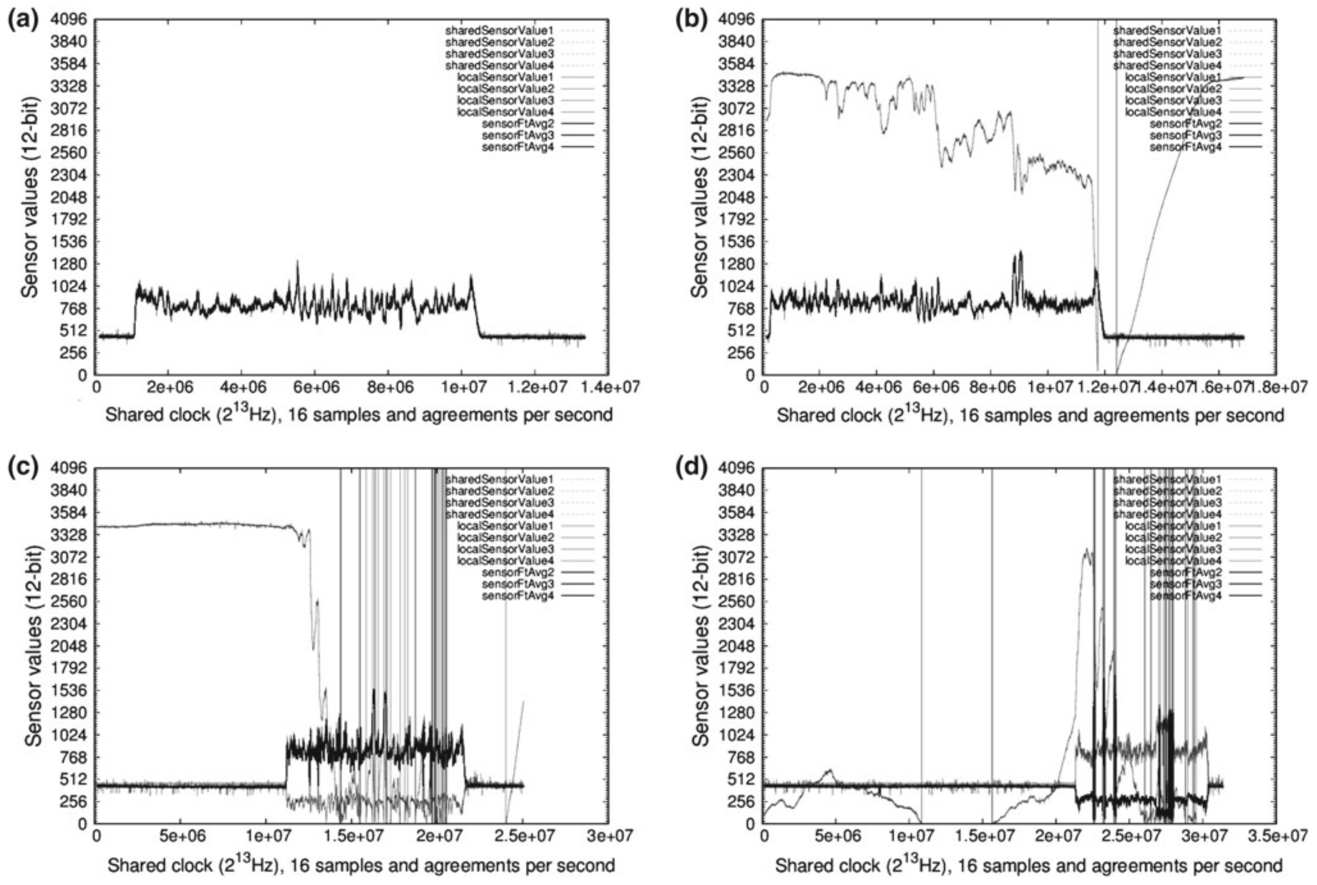

Fig. 11 Logged pressure sensor, voted and averaged data

the scenarios are the number of physical faults. In Fig. 11a, no physical faults are introduced; in Fig. 11b, one benign fault has been injected by putting a cap over the total pressure probe of one local tube.[7] In Fig. 11c, in addition to the capped tube, adhesive tape is placed over another tube, and in Fig. 11d, the tape is placed over two tubes in addition to the capped tube.

The plots depict the pressure difference samples 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 Fig. 11d, 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 Fig. 11b, 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 functions of the stuck tube result in an underflowing value. In Fig. 11d, 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.

Fault-tolerant monitoring of real-time systems can relieve the monitored underlying implementation of adding in and executing fault-tolerant variants of the executed algorithms (but does not exclude double-checking such). It can be used as a way to simplify the software assurance effort, while still having fault-tolerance separately added through monitors.

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. Although 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 ground-station software packages, like QGroundControl, Happy Killmore Ground Control Station, the Ardupilot Mega Planner and autopilot systems like

[7] Tape left on the static pitot tube of Aeroperú Flight 603 in 1996 resulted in the death of 70 passengers and crew [28].

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

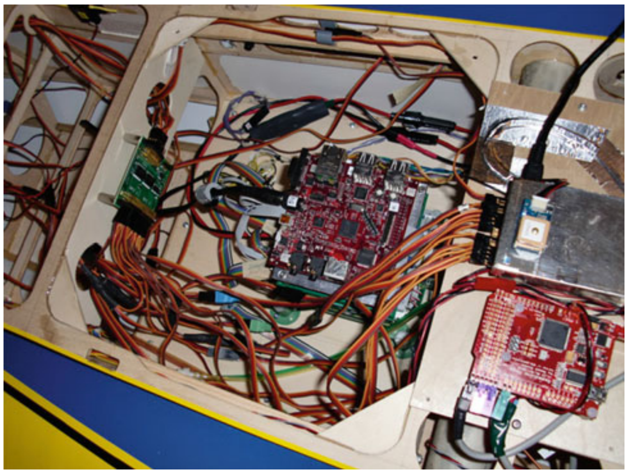
Fig. 12 Beagle Board executing the MAVLink monitor

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 MAVLink packets from an Ardupilot Mega. The configuration in the Edge is depicted in Fig. 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 ZigBee 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 go out of reception 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' = startSequenceSize :: Stream Word32

headerSize  = 6
headerSize' = headerSize :: Stream Word32
```

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

```
crcSize  = 2
crcSize' = crcSize :: Stream Word32
```

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 values and payload by defining a stream for each field, dropping values from the `inputStream` according to Table 1.

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

[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.

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

where `.&.` and `.^.` denote bitwise `and`, and 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` function returns a Haskell list of Copilot streams. A stream at position *i* in the list calculates the CRC value over the prefix of length *i* of the `inputStream`, excluding the possible start sign.

Since Copilot is a synchronous language, with each new input byte sampled from the C program in a tick, all stream values will be recalculated. In each tick, we can check if we have a possible valid packet candidate. The CRC of a valid packet in the `crcStreams` list at the position after a suspected packet CRC will be zero:

```
crcIndex :: Stream Word32
crcIndex = headerSize' + payloadLength
              - 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, a boolean stream that signals valid packet candidates can be given by two more definitions [11]:

```
startMatch :: Stream Bool
startMatch  = inputStream == 0x55

validPacket :: Stream Bool
validPacket = startMatch && crcValid
```

[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, ...]`.

[11] We could incorporate further analysis of the packets as well, like checking for the correct length of certain MAVLink packet types or inspection of the payload. Some of these tests could be derived from the MAVLink XML protocol description automatically.

The communication between an autopilot and a groundstation 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. 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 signals True from start to the end
-- of a valid packet
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 ) )
```

The `analyzingPacket` stream uses a helper stream `analyzingPacket'` as a counter, `analyzingPacket` stays true as long as the counter is greater zero. If we are not yet analyzing a packet and the `validPacket` signal becomes true, the counter is set to the value of the packet length field plus header and field CRC sizes. In any other case, the counter decrements until zero on each tick.

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 several 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:

```
packetWithID mId = validPacket && messageID == mId

packetWithIDLength mId pLen =
    packetWithID mId && payloadLength == pLen

-- the global position in integer values has
-- message id 73  and payload length 18
globalPositionINT = packetWithIDLength 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 [12]. Reconstruction of the position is done by three 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 that takes two streams as parameters.

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 Figs. 13 and 14, respectively, which graph the latitude, longitude, and altitude of the aircraft during two flights. Comparing the graphs, in Fig. 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 for avionics monitoring. In the first case-study, Copilot-generated monitors are used to implement fault-tolerance mechanisms to decompose the problem of implementing a sensor system and fault-tolerance. In the second, Copilot allows us to synthesize protocol parsers and analyzers.

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.[13] The Copilot monitor generates about 2,500 lines of real-time C code.

7 Related work

Copilot shares similarities with other RV systems that emphasize real-time or distributed systems. Krüger et al. [27] describe their work in synthesizing monitors for a automobile door-locking system. While the system is distributed, it is not ultra-reliable and is not hard real-time or fault-tolerant via hardware replication. The implementation is in Java and focuses on the aspect-oriented monitor synthesis, similar in spirit to JavaMOP [12]. SYNCRAFT is a tool that takes a distributed program (specified in a high-level modeling language) that is fault-intolerant and given some invariant and fault model, transforms the program into one that is fault-tolerant (in the same modeling language) [8].

There are few instances of RV focused on C code. One exception is RMOR, which generates constant-memory C monitors [21]. RMOR does not address real-time behavior or distributed system RV, though.

Research at the University of Waterloo also investigates the use of time-triggered RV (i.e., periodic sampling). Unlike with Copilot, the authors do not make the assumptions that the target programs are hard real-time themselves, so a significant portion of the work is devoted to developing the theory of efficiently monitoring for state changes using

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

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

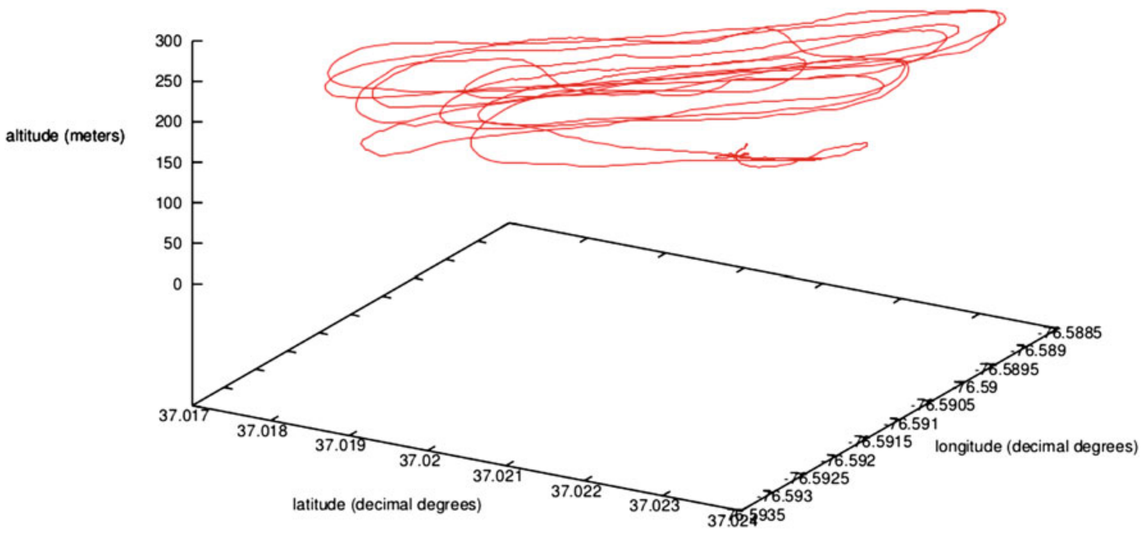

Fig. 13 Positional data for flight 1

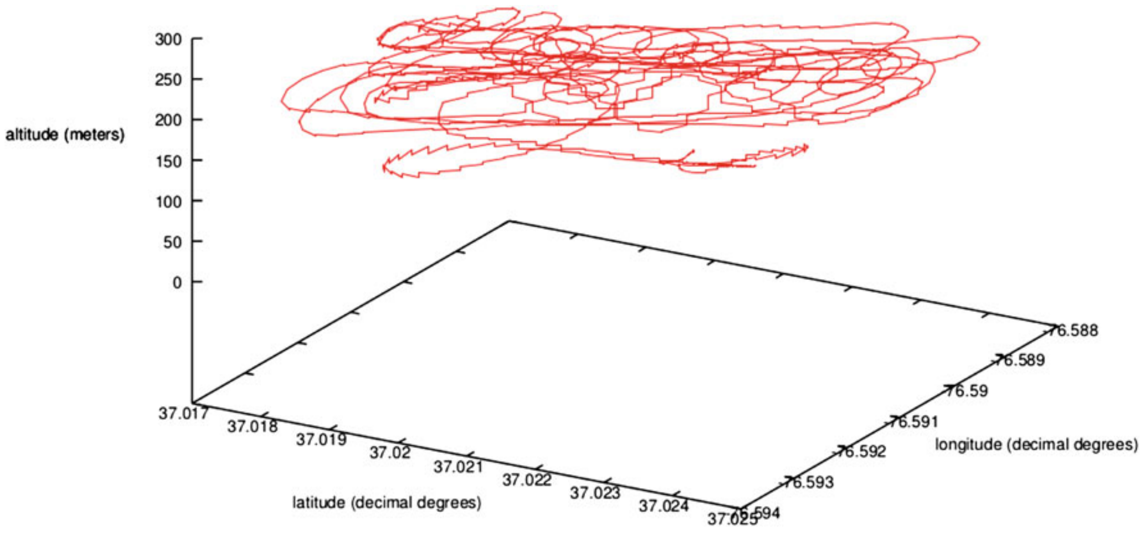

Fig. 14 Positional data for flight 2

time-triggered RV for arbitrary programs, particularly for testing [9, 18]. On the other hand, the work does not address issues such as distributed systems, fault-tolerance, or monitor integration.

With respect to work outside of RV, other research also addresses the use of eDSLs for generating embedded code. Besides Atom [22], which we use as a back-end, Feldspar is an eDSL for digital signal processing [5]. Copilot is similar in spirit to other languages with stream-based semantics, notably represented by the Lustre family of languages [32]. Copilot is a simpler language, particularly with respect to Lustre's clock calculus, focused on monitoring (as opposed to developing control systems). Copilot can be seen as an generalization of the idea of Lustre's "synchronous observers" [20], which are Boolean-valued streams used to track properties about Lustre programs. Whereas Lustre uses synchronous observers to monitor Lustre programs, we apply the idea to monitoring arbitrary periodically scheduled real-time systems. The main advantages of Copilot over Lustre is that Copilot is implemented as an eDSL, with the associated benefits; namely Haskell compiler and library reuse the ability to define polymorphic functions, like the `majority` macro in Sect. 4.7 that gets monomorphized at compile time.

8 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.

8.1 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 Sect. 4.7 is compiled into a single Copilot expression. The use of explicit sharing (Sect. 4.6) reduces the cost of computation by ensuring subexpressions are not needlessly recomputed, but if the subexpressions themselves are expensive to compute, the entire expression becomes expensive. This is analogous to a standard compiler inlining *every* function, which would result in infeasible code-size.

Techniques to improve the efficiency of evaluating embedded domain-specific languages (eDSLs) and to transfer “expression sharing” from the host language to the DSL language are needed. Fortunately in our domain, monitoring code is terse, in general.

8.2 Assurance

The lightweight approach to monitor assurance discussed in Sect. 5 is described in more detail in [36]. The current framework for front-end/back-end testing is built on Quickcheck and model-checking, which does not provide coverage testing capability. Given the criticality of the monitors, DO-178 [37] would require that MC/DC testing be performed on them if they were to be employed in industrial avionics. Adding this capability to the toolchain is a challenge for future work.

8.3 Scheduling monitors

In the experiments described in Sect. 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 easier 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 [2,3] 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 dependent. It would help the specifier to lift the abstraction level, so she can write properties in terms of real-time.

8.4 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.

8.5 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.

8.6 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 [40].

8.7 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.

Acknowledgments This work was supported by NASA Contract NNL08AD13T. Portions of this work have been published as conference papers in *Runtime Verification*, 2010 and *Runtime Verification*, 2011. We wish to especially thank the following individuals: Ben Di Vito at the NASA Langley Research Center (NASA LaRC) monitored this contract, and Paul Miner and Eric Cooper, also at NASA LaRC, provided valuable input. Robin Morisset developed an earlier version of Copilot.

References

1. (2000) FAA system handbook. http://www.faa.gov/library/manuals/aviation/risk_management/ss_handbook/
2. (2010) Aeronautical radio: avionics application software standard interface: ARINC specification 653p1-3. ARINC, Inc., Annapolis. ARINC 653 Part 1
3. (2012) Aeronautical radio: avionics application software standard interface: ARINC specification 653p2-2 extended services. ARINC Inc., Annapolis. ARINC 653 Part 2
4. (2011) Aviation Today: more pitot tube incidents revealed. Aviation Today. http://www.aviationtoday.com/regions/usa/More-Pitot-Tube-Incidents-Revealed_72414.html
5. Axelsson E, Claessen K, Dévai G, Horváth Z, Keijzer K, Lyckegård B, Persson A, Sheeran M, Svenningsson J, Vajda A (2010) Feldspar: a domain specific language for digital signal processing algorithms. In: 8th ACM/IEEE international conference on formal methods and models for codesign
6. Barrett C, Sebastiani R, Seshia S, Tinelli C (2009) Satisfiability modulo theories, chap. 26, pp 825–885. In: Frontiers in artificial intelligence and applications. IOS Press, Amsterdam
7. Bergin C (2008) Faulty MDM removed. NASA Spaceflight.com. http://www.nasaspaceflight.com/2008/05/sts-124-frr-debate-outstanding-issues-faulty-mdm-removed/. Downloaded 28 Nov 2008
8. Bonakdarpour B, Kulkarni SS (2008) SYCRAFT: a tool for synthesizing distributed fault-tolerant programs. In: International conference on concurrency theory (CONCUR '08). Springer, Berlin, pp 167–171
9. Bonakdarpour B, Navabpour S, Fischmeister S (2011) Sampling-based runtime verification. In: 17th International symposium on formal methods (FM)
10. Bureau ATS (2007) 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. Aviation Occurrace Report-200503722
11. Butler RW, Finelli GB (1993) The infeasibility of quantifying the reliability of life-critical real-time software. IEEE Trans Softw Eng 19:3–12
12. Chen F, d'Amorim M, Roşu G (2006) Checking and correcting behaviors of java programs at runtime with Java-MOP. Electron Notes Theor Comput Sci 144:3–20
13. Chen F, Roşu G (2005) Java-MOP: a monitoring oriented programming environment for Java. In: 11th International conference on tools and algorithms for the construction and analysis of systems (TACAS'05). LNCS, vol 3440. Springer, Berlin, pp 546–550
14. Claessen K, Hughes J (2000) Quickcheck: a lightweight tool for random testing of haskell programs. In: ACM SIGPLAN notices. ACM, New York, pp 268–279
15. Clarke E, Kroening D, Lerda F (2004) A tool for checking ANSI-C programs. In: Tools and algorithms for the construction and analysis of systems (TACAS). LNCS. Springer, Berlin, pp 168–176
16. Dwyer M, Diep M, Elbaum S (2008) Reducing the cost of path property monitoring through sampling. In: Proceedings of the 23rd international conference on automated software engineering, pp 228–237
17. Farhat H (2004) Digital design and computer organization, 1st edn. In: Digital Design and Computer Organization. CRC Press, Boca Raton
18. Fischmeister S, Ba Y (2010) Sampling-based program execution monitoring. In: ACM International conference on Languages, compilers, and tools for embedded systems (LCTES), pp 133–142
19. Gill A (2009) Type-safe observable sharing in Haskell. In: Proceedings of the 2009 ACM SIGPLAN Haskell Symposium
20. Halbwachs N, Raymond P (1999) Validation of synchronous reactive systems: from formal verification to automatic testing. In: ASIAN'99 Asian computing science conference. LNCS, vol 1742. Springer, Berlin
21. Havelund K (2008) Runtime verification of C programs. In: Testing of software and communicating systems (TestCom/FATES). Springer, Berlin, pp 7–22
22. Hawkins T (2008) Controlling hybrid vehicles with Haskell. Presentation. Commercial Users of Functional Programming (CUFP). http://cufp.galois.com/2008/schedule.html
23. Hesselink WH (2005) The Boyer–Moore majority vote algorithm
24. Jones SP (ed) (2002) Haskell 98 language and libraries: the revised report. http://haskell.org/
25. Kim M, Viswanathan M, Ben-Abdallah H, Kannan S, Lee I, Sokolsky O (1999) Formally specified monitoring of temporal properties. In: 11th euromicro conference on real-time systems, pp 114–122
26. 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 (2010) seL4: Formal verification of an OS kernel. Commun ACM 53(6):107–115
27. Krüger IH, Meisinger M, Menarini M (2007) Runtime verification of interactions: from MSCs to aspects. In: International conference on runtime verification. Springer, Berlin, pp 63–74
28. Ladkin PB (2002) News and comment on the Aeroperu b757 accident; AeroPeru Flight 603, 2 October 1996. Online article RVS-RR-96-16. http://www.rvs.uni-bielefeld.de/publications/Reports/aeroperu-news.html
29. Lamport L, Shostak R, Pease M (1982) The Byzantine generals problem. ACM Trans Program Lang Syst 4:382–401
30. Leveson NG, Turner CS (1993) An investigation of the Therac-25 accidents. Computer 26:18–41
31. Macaulay K (2008) 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. http://www.atsb.gov.au/newsroom/2008/release/2008_45.aspx
32. Mikác J, Caspi P (2005) Formal system development with Lustre: framework and example. Technical Report TR-2005-11, Verimag Technical Report. http://www-verimag.imag.fr/index.php?page=techrep-list&lang=en
33. Moore SJ, Boyer RS (1981) MJRTY—a fast majority vote algorithm. Technical Report 1981-32, Institute for Computing Science, University of Texas

34. Nuseibeh B (1997) Soapbox: Ariane 5: Who dunnit? IEEE Softw 14(3):15–16
35. Pike L, Goodloe A, Morisset R, Niller S (2010) Copilot: a hard real-time runtime monitor. In: Runtime verification (RV), vol 6418. Springer, Berlin, pp 345–359
36. Pike L, Wegmann N, Niller S, Goodloe A (2012) Experience report: do-it-yourself high-assurance compiler. In: Proceedings of the 17th ACM SIGPLAN conference on functional programming. ACM, New York
37. RTCA (1992) Software considerations in airborne systems and equipment certification. RTCA, Inc., USA. RCTA/DO-178B
38. Rushby J (2008) Runtime certification. In: RV'08: Proceedings of runtime verification, Budapest, Hungary, March 30, 2008. Selected Papers. Springer, Berlin, pp 21–35
39. Rushby J (2009) Software verification and system assurance. In: International conference on software engineering and formal methods (SEFM). IEEE, New York, pp 3–10
40. Sammapun U, Lee I, Sokolsky O (2005) 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
41. Stoller SD, Bartocci E, Seyster J, Grosu R, Havelund K, Smolka SA, Zadok E (2011) Runtime verification with state estimation. In: Proceedings of the 2nd international conference on runtime verification (RV'11)