Copilot: Monitoring Embedded Systems Final Report[*]

Lee Pike
Galois, Inc.
leepike@galois.com

Nis Wegmann
University of Copenhagen
wegmann@diku.dk

Sebastian Niller
National Institute of Aerospace
sebastian.niller@gmail.com

Alwyn Goodloe
NASA Langley Research Center
a.goodloe@nasa.gov[†]

November 2, 2011

Abstract

Runtime verification (RV) is a natural fit for ultra-critical systems, where correctness is imperative. In ultra-critical systems, even if the software is fault-free, because of the inherent unreliability of commodity hardware and the adversity of operational environments, processing units (and their hosted software) are replicated, and fault-tolerant algorithms are used to compare the outputs. We investigate both software monitoring in distributed fault-tolerant systems, as well as implementing fault-tolerance mechanisms using RV techniques. We describe the Copilot language and compiler, specifically designed for generating monitors for distributed, hard real-time systems. We also describe two case-studies in which we generated Copilot monitors in avionics systems.

[*]This work is supported by NASA Contract NNL08AD13T. Portions of this report have been published as a conference paper [1].

[†]The research contributed by Alwyn Goodloe was performed during his tenure as a staff scientist at the National Institute of Aerospace.

Contents

List of Figures

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 [2]. The justification for the requirement is essentially that for reasonable estimates for the size of an aircraft fleet, the number of hours of operation per aircraft in its lifetime, and the number of critical aircraft subsystems, a 10^{-9} probability of failure per hour ensures that the overall probability of failure for the aircraft fleet is "sufficiently small." 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 functional correctness of approximately ten thousand lines of code represents the state-of-the-art [3]. Nearly 20 years ago, Butler and Finelli showed that testing alone cannot verify the reliability of ultra-critical software [4].

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. Furthermore, 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 sensing and/or controlling some physical plant and that they are *hard real-time*, meaning that deadlines are fixed and time-critical.

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 [5] and the Ariane 5 Flight 501 disaster [6]. 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 drawing 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 [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 [8]), 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 [9]. 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. Due to bugs in the software, accelerometer A's failure was never reported so maintenance could be 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) [10]. 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 first present constraints to runtime monitoring for real-time embedded systems, then we present Copilot, our approach for satisfying these constraints.

3.1 RV Constraints

Ideally, the RV approaches that have been developed in the literature could be applied straightforwardly to ultra-critical systems. Unfortunately, these systems have constraints violated by typical RV approaches. 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 [11]) 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) [12] and Monitor Oriented Programming (MOP) [13] 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 require the most significant deviations from traditional RV approaches. We have previously argued 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 [14]. Other work at the University of Waterloo also investigates time-triggered RV [15, 16].

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

3.2 Preliminaries

Copilot is embedded into the functional programming language Haskell [17]. A working knowledge of Haskell is necessary to use Copilot effectively; a variety of books and free web resources introduce Haskell. Copilot uses Haskell language extensions specific to the Glasgow Haskell Compiler (GHC); hence in order to start using Copilot, you must first install an up-to-date version of GHC. (The minimal required version is 7.0.) The easiest way to do this is to download and install the Haskell Platform, which is freely distributed from here:

http://hackage.haskell.org/platform

After having installed the Haskell Platform, Copilot is downloaded and installed by executing the following command:

```
> cabal install copilot
```

This should, if everything goes well, install Copilot on your system.

Copilot is distributed throughout a series of packages at Hackage:

- copilot-language: Contains the language front-end.
- copilot-core: Contains an intermediate representation for Copilot programs (shared by all back-ends).
- copilot-c99: A backend for Copilot targeting C99 (based on Atom, http://hackage.haskell.org/package/atom).
- copilot-sbv: A backend for Copilot targeting C99 (based on SBV, http://hackage.haskell.org/package/sbv).
- copilot-libraries: A set of utility functions for Copilot, including a clock-library, a linear temporal logic framework, a voting library, and a regular expression framework.
- copilot-cbmc: A driver for proving the correspondence between code generated by the copilot-c99 and copilot-sbv backends.

Many of the examples in this paper can be found at https://github.com/leepike/Copilot/tree/copilot2.0/Examples.

3.3 Domain

Copilot is a domain-specific language tailored to programming *runtime monitors* for *hard real-time*, *distributed*, *reactive systems*. Briefly, a runtime monitor is program that runs concurrently with a target program with the sole purpose of assuring that the target program behaves in accordance with a pre-established specification. Copilot is a language for writing such specifications.

A reactive system is a system that responds continuously to its environment. All data to and from a reactive system is communicated progressively during execution. Reactive systems differ from transformational systems which transforms data in a single pass and then terminate, as for example compilers and numerical computation software.

A hard real-time system is a system that has a statically bounded execution time and memmory usage. Typically, hard real-time systems are used in mission-critical software, such as avionics, medical equipment, and nuclear power plants; hence, occasional dropouts in the response time or crashes are not tolerated.

A distributed system is a system which is layered out on multiple pieces of hardware. The distributed systems we consider are all synchronized, i.e., each component agree on a shared global clock.

3.4 Language

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

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

```
x :: Stream Int32        y :: Stream Int32        z :: Stream Int32
x = 5 + 5                y = x * x                z = x == 10 && y < 200
```

Here the streams `x`, `y`, and `z` are simply *constant streams*:

$$\texttt{x} \rightsquigarrow \{10, 10, 10, \dots\},\ \texttt{y} \rightsquigarrow \{100, 100, 100, \dots\},\ \texttt{z} \rightsquigarrow \{\mathrm{T},\ \mathrm{T},\ \mathrm{T},\ \dots\}$$

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
```

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

```
u = drop 2 w
```

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

3.4.1 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
```

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:

```
h :: Stream Word64
h = drop 2 g
  where g = f
        f = [0,1,2] ++ f
```

But if instead `h` is defined as `h = drop 4 g`, 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 `g` and `f` to be generated before producing values for `h`. This is not possible since Copilot programs may take inputs in real-time from the environment (see Section 3.4.4).

3.4.2 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
    where
    z = [False] ++ y
```

Figure 1: A latch. 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.*

3.4.3 Stateful Functions

In addition to pure functions, such as `even` and `mul_comp`, Copilot also facilitates *stateful* functions. A *stateful* function is 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 [18], 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 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* $\texttt{cnt}_i$ *is determined as shown in the left table in Figure 2.*

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

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

```
counter :: Stream Bool -> Stream Bool
        -> Stream Int
counter inc reset = cnt
  where
    cnt | reset     = 0
        | inc       = z + 1
        | otherwise = z
    z = [0] ++ cnt
```

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

3.4.4 Interacting With the Target Program

All interaction with the outside world is done by sampling *external variables* and by evoking *triggers*. External variables are variables that are defined outside Copilot and which reflect the visible state of the target program that we are monitoring. 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.

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

```
extern :: Typed a => String -> Stream a
```

It takes the name of an external variable and generates a stream by sampling the variable at each clock cycle.

Triggers 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 = extern "x" :: Stream Int32
  trigger "h" (x < 10) [arg x]
```

The order in which the triggers are defined is irrelevant.

Example 5:
We consider an engine controller with the following property: If the temperature rises more than 2.3 degrees within 0.2 seconds, then the engine should be shut off immediately. Assuming that the global samplerate is 0.2 seconds, we can define a monitor that surveys the above property:

```
propTempRiseShutOff :: Spec
propTempRiseShutOff = trigger "over_temp_rise" (overTempRise ==> not running) []
  where
  temps        = [0, 0, 0] ++ (extern "temp" :: Stream Float)
  overTempRise = drop 2 temps > const 2.3 + temps
  running      = extern "running"
```

Here, we assume that the external variable `temp` *denotes the temperature of the engine and the external variable* `running` *indicates whether the engine 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.*

In addition to external variables, we can also sample external arrays and functions:

```
externArray :: (Typed a, Typed b, Integral a) => String -> Stream a -> Stream b
externFun :: Typed a => String -> [FunArg] -> Stream a
funArg :: Typed a => Stream a -> FunArg
```

The constructor `externArray` takes two arguments: the name of array and an index into the array. The index is given by a Copilot stream (of integral type) that is used as an index into the table. The constructor `externFun` also takes two arguments: the name of the external function and a list of arguments that are provided to the function. Similarly to the index for external arrays, each argument to an external function is given by a Copilot stream. 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.

Example 6:
Say we have defined a lookup-table (in C99) of a discretized continuos function that we want to use within Copilot:

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

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

```
lookupSomeTable :: Stream Int -> Stream Stream Double
lookupSomeTable k = externArray "someTable" k
```

Given the following values for `k`*,* $\{2, 1, 3, 3, 2, \ldots\}$*, the output of* `lookupSomeTable k` *would be* $\{3.7, 3.5, 4.5, 4.5, 3.7, \ldots\}$.

3.4.5 Explicit Sharing

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

Figure 3: Implicit sharing (to the left) versus explicit sharing (to the right).

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 ordinary Haskell. From a semantic point of view the streams `s1` and `s2` from Figure 3 are identical. As we will see in Section 3.6, however, certain advanced Copilot programs may force the compiler to build syntax trees that blow up exponentially. In such cases, using explicit sharing may help to avoid this.

3.5 Tools

Copilot comes with a variety of tools, including a pretty-printer, an interpreter, a two compilers targeting C, and a verifier front-end. In the following section, we will demonstrate some of these tools and their usage.

3.5.1 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 3.6) can generate expressions that are quite large. Very large expressions can take significant time to interpret or compile.

3.5.2 Interpreting Copilot

Suppose we want to interpret the engine controller from Example 5. Before we can do so, we must define the external variables `temp` and `running` as infinite Haskell lists:

```
temp :: [Float]
temp = map exp [0.2, 0.4 ..]

running :: [Bool]
running = repeat True
```

We now evoke the interpreter as follows (e.g. in GHCI, the GHC compiler's interpreter for Haskell):

```
interpret
  10
  [input "temp" temp, input "running" running]
  propTempRiseShutOff
```

The first argument to the function *interpret* is the number of iterations that we want to evaluate. The second argument is a list of inputs. Each input takes a name and an infinite list of values (which must conform to the type of the input). The third argument is the specification that we wish to interpret.

3.5.3 Compiling Copilot

Compiling the engine controller from Example 5 is straightforward. First, we pick a backend to compile to (see Section 3.2 for a brief list of backends; we compile to Atom below), import it, and compile as follows:[2]

```
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" —

[2]Two explanations are in order: (1) `reify` is an optimization that improves sharing in the expressions to be compiled, improving efficiency [19], and `>>=` is a higher-order operator that takes the result of reification and "feeds" it to the compile function.

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"

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!
```

3.5.4 QuickCheck

QuickCheck [20] 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 backend 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.

If you have installed Copilot, you can execute the quickCheck tests by executing the program `CopilotC99Test`. The default installation for the executable is in `$HOME/.cabal/bin`. Assuming the executable is in your path, simply execute it. It will direct the user to enter the number of specifications to test. The program will then generate that many random specifications, testing the output of the interpreter against the executed C program. By default, it tests the outputs for ten iterations.

3.5.5 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 backends that each generate C code (similar drivers are planned for future backends). 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 backend and one through the copilot-sbv backend. In addition, a driver (that is, a `main` function) is generated that executes the code from each backend. 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 [21]. A license for the tool is available. [3] CBMC must be downloaded and installed separately;

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

```
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'
```

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

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 backends 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
```

3.6 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 there in a given list of votes is a candidate that has more than half of the votes, and if so, of finding this candidate.

```
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
```

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

The Boyer-Moore Majority Vote Algorithm [22, 23] solves the problem in linear time and constant memory. It does so in two passes: The first pass eliminates every, but a single, candidate; and the second pass asserts that the found candidate indeed holds a majority.

Without going into details of the algorithm (references are provided in the bibliography), 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.

```
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'
```

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

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 in reality 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 build-in *let*-blocks, we use explicit sharing,

```
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
```

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

as described in Section 3.4.5. 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.

4 Case Studies: Monitoring Avionics

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

4.1 Pitot Tube Fault-Tolerance

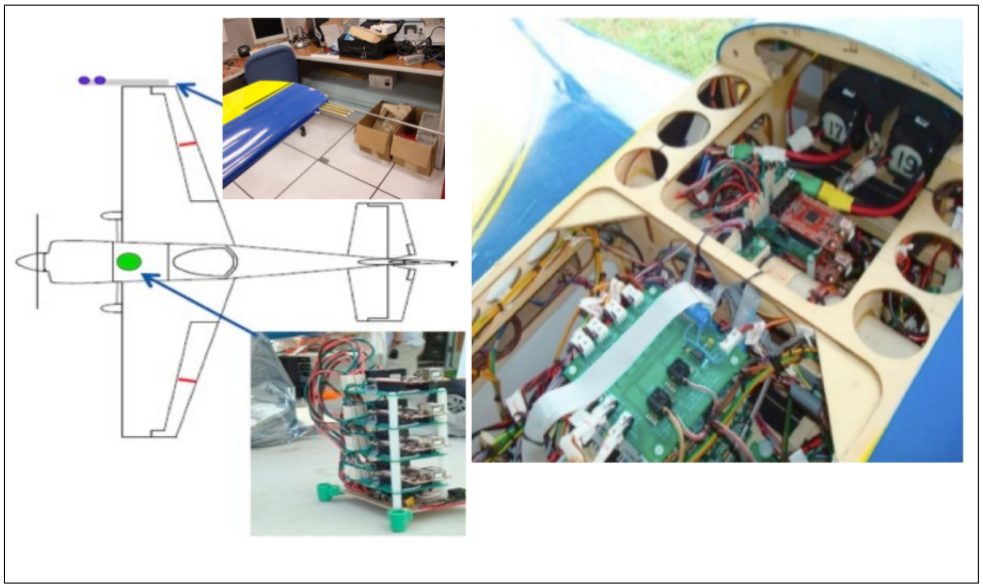


Figure 8: Stack configuration in the Edge 540 aircraft.

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 [24], 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 the Edge 540 test aircraft, is depicted in Figure 8.

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

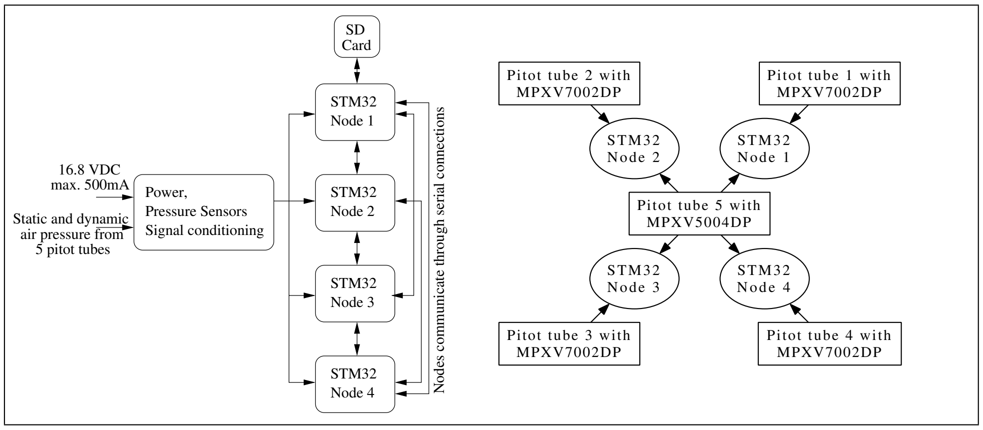

SD Card
STM32 Node 1
STM32 Node 2
STM32 Node 3
STM32 Node 4
16.8 VDC max. 500mA
Static and dynamic air pressure from 5 pitot tubes
Power, Pressure Sensors Signal conditioning
Nodes communicate through serial connections
Pitot tube 2 with MPXV7002DP
Pitot tube 1 with MPXV7002DP
Pitot tube 5 with MPXV5004DP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

Figure 9: Hardware stack and pitot tube configuration.

(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 9). 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 [8]). 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.

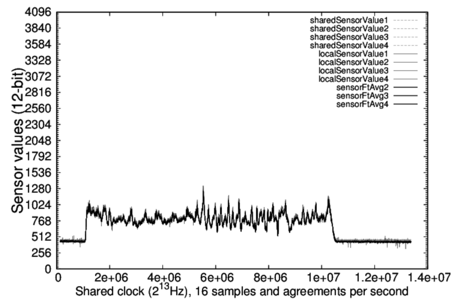
3. Each monitor performs a majority vote (as described in Section 3.6) over the three values it has for every other monitor of the shared sensor (call this $maj_i(S)$ for node i) and the local sensor (call this $maj_i(L)$ for node i).

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.

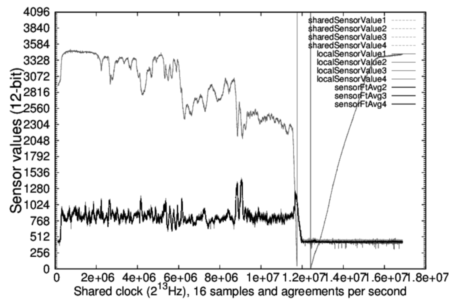
5. Another fault-tolerant average is taken over a five-element set, where the two least and two greatest elements are removed (thus returning the median value). The set contains the fault-tolerant average over the shared sensor described in the previous step ($ftAvg(S)$), the node's local sensor value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

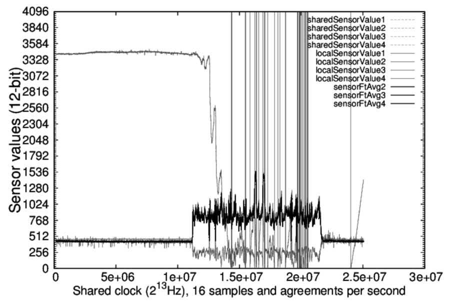
The graphs in Figure 10 depict four scenarios in which different faults are injected. In each scenario, there is a software-injected Byzantine faulty node present. What varies between the scenarios are the number of physical faults. In Figure 10(a), no physical faults are introduced; in Figure 10(b), one benign fault has been injected by putting a cap over the total pressure



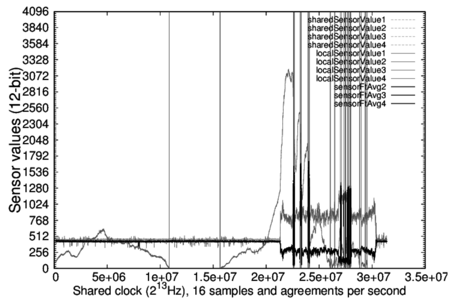
(a)



(b)



(c)



(d)

Figure 10: Logged pressure sensor, voted and averaged data.

probe of one local tube.[4] In Figure 10(c), in addition to the capped tube, sticky tape is placed over another tube, and in Figure 10(d), sticky tape is placed over two tubes in addition to the capped tube.

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 10(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 10(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

[4]Tape left on the static pitot tube of Aeroperú Flight 603 in 1996 resulted in the death of 70 passengers and crew [25].

underflowing value. In Figure 10(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.

4.2 MAVLink Monitoring

The MAVLink (Micro Air Vehicle Link[5]) protocol consists of a set of messages to be sent between small air vehicles and ground stations. Althought it can be used on human-controlled vehicles to send report messages on parameters like wind speed or attitude, the usual applications of MAVLink are in avionic systems with an autopilot. MAVLink is used by several ground-station 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.

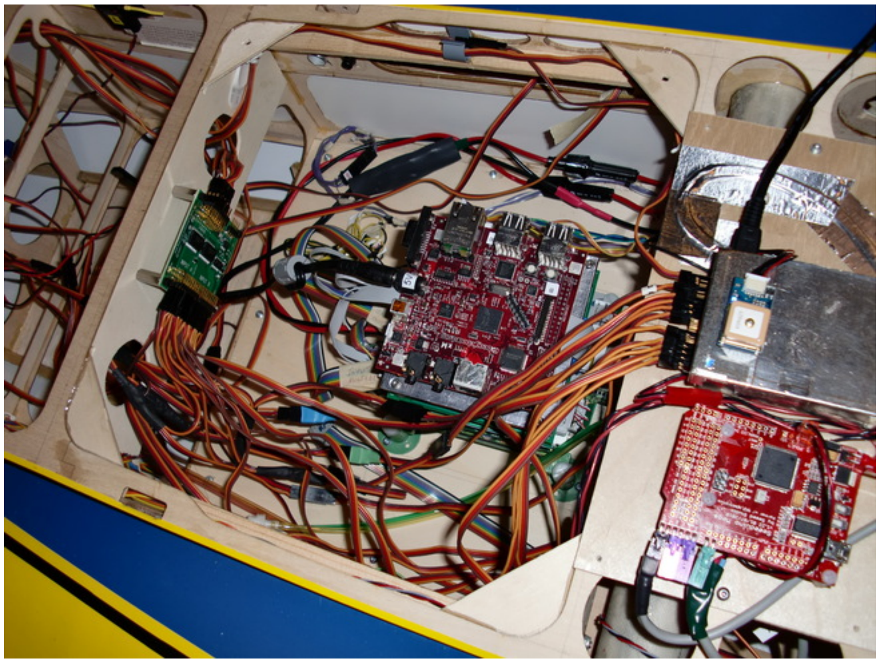


Figure 11: Beagle Board executing the MAVLink monitor.

We have implemented portions of the common set MAVLink protocol as Copilot monitors and have executed them on binary MAVLink log files.

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

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

Table 1: MAVLink packet fields.

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 Figure 11. 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 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.[6] In each tick, the next MAVLink byte is sampled from the C varible `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"
```

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
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 bit 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' ]
```

[6] 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.

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.[7]

```
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 - 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 appaer 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:

[7]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, ...]`.

```
-- 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 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:

```
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 [8].

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

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
```

The streams become parameters of a globalPositionInt trigger:

```
trigger "globalPositionINT" globalPositionINT [ 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 12 and 13, respectively, which graph the latitude, longitude, and altitude of the aircraft during two flights. Comparing the graphs, in Figure 13, 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

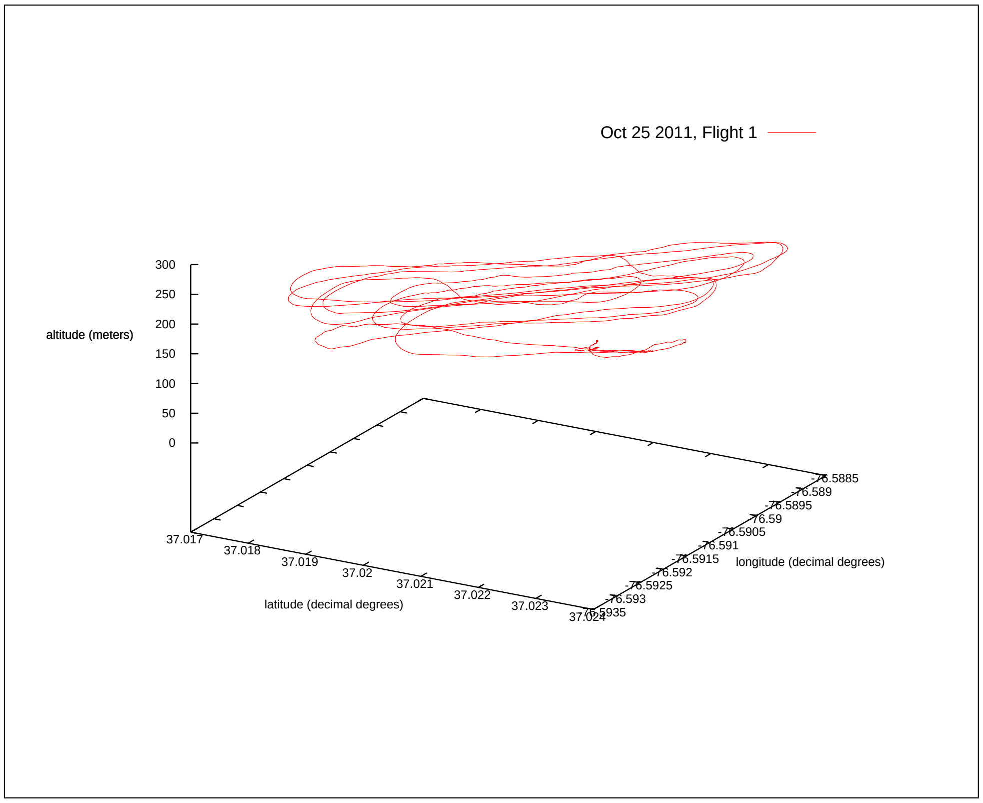


Figure 12: Flight 1.

with the changed altitude (the altitude changes because–while GPS altitude are not updated–barometric altitude is) and causes the stair effect.

4.3 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.[9] The Copilot monitor generates about

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

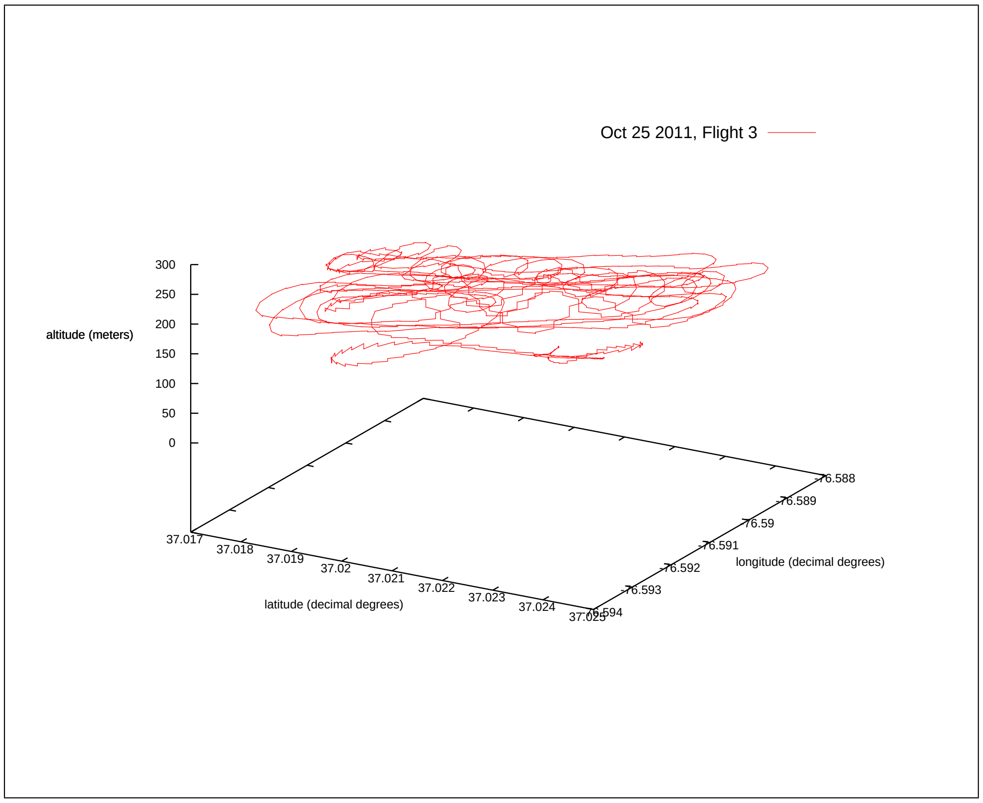


Figure 13: Flight 2.

2500 lines of real-time C code.

5 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.

eDSL efficiency. First, we have demonstrated that the embedded DSL approach is quite powerful, turning regular programming on its head: while Copilot is quite simple, its macro language is a higher-order functional language! One disadvantage of this approach is that particularly with a powerful macro language, it is easy to build up very large expressions—much larger than would be built in a conventional programming language. For example, the Boyer-Moore voting algorithm described in Section 3.6 is compiled into a single Copilot expression. The use of explicit sharing (Section 3.4.5) 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 relatively terse, in general.

Scheduling monitors. In the experiments described in Section 4, 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).

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] ++ externI32 "x"
  x' = drop 1 x
```

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.

Other language features. In analyzing protocol streams, reconstructing values of 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 envelop.

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 [26].

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.

Acknowledgements

The authors are grateful to NASA for Contract NNL08AD13T. In particular, 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] Lee Pike, Sebastian Niller, and Nis Wegmann. Runtime verification for ultra-reliable systems. In *Proceedings of the 2nd Intl. Conference on Runtime Verification*, 2011. 1

[2] John Rushby. Software verification and system assurance. In *Intl. Conf. on Software Engineering and Formal Methods (SEFM)*, pages 3–10. IEEE, November 2009. 4

[3] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM (CACM)*, 53(6):107–115, June 2010. 4

[4] R. W. Butler and G. B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19:3–12, January 1993. 4

[5] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26:18–41, 1993. 4

[6] Bashar Nuseibeh. Soapbox: Ariane 5: Who dunnit? *IEEE Software*, 14(3):15–16, 1997. 4

[7] Chris Bergin. Faulty MDM removed. NASA Spaceflight.com, May 18 2008. Available at http://www.nasaspaceflight.com/2008/05/sts-124-frr-debate-outstanding-issues-faulty-mdm-removed/. (Downloaded Nov 28, 2008). 5

[8] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, July 1982. 5, 24

[9] Australian Transport Safety Bureau. 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. 5

[10] Kerryn Macaulay. 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, November 14 2008. Available at http://www.atsb.gov.au/newsroom/2008/release/2008_45.aspx. 5

[11] RTCA. Software considerations in airborne systems and equipment certification. RTCA, Inc., 1992. RCTA/DO-178B. 6

[12] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *11th Euromicro Conference on Real-Time Systems*, pages 114–122, 1999. 6

[13] F. Chen and G. Roşu. 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)*, volume 3440 of *LNCS*, pages 546–550. Springer, 2005. 6

[14] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A hard real-time runtime monitor. In *Runtime Verification (RV)*, volume 6418, pages 345–359. Springer, 2010. 7

[15] S. Fischmeister and Y. Ba. Sampling-based program execution monitoring. In *ACM International conference on Languages, compilers, and tools for embedded systems (LCTES)*, pages 133–142, 2010. 7

[16] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. Sampling-based runtime verification. In *17th Intl. Symposium on Formal Methods (FM)*, 2011. 7

[17] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, 2002. 8

[18] H.A. Farhat. *Digital design and computer organization*. Number v. 1 in Digital Design and Computer Organization. CRC Press, 2004. 12

[19] Andy Gill. Type-safe observable sharing in Haskell. In *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, September 2009. 16

[20] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN Notices*, pages 268–279. ACM, 2000. 18

[21] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and*

Analysis of Systems (TACAS), LNCS, pages 168–176. Springer, 2004. 19

[22] Strother J. Moore and Robert S. Boyer. MJRTY - A Fast Majority Vote Algorithm. Technical Report 1981-32, Institute for Computing Science, University of Texas, February 1981. 21

[23] Wim H. Hesselink. The boyer-moore majority vote algorithm, 2005. 21

[24] Aviation Today. More pitot tube incidents revealed. Aviation Today, February 2011. Available at http://www.aviationtoday.com/regions/usa/More-Pitot-Tube-Incidents-Revealed_72414.html. 23

[25] Peter B. Ladkin. 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. 26

[26] U. Sammapun, I. Lee, and O. Sokolsky. 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*, pages 147–153, 2005. 36