Appears in the proceedings of the *2nd International Conference on Runtime Verification* (RV 2011).

Runtime Verification for Ultra-Critical Systems

Lee Pike[1], Sebastian Niller[2], and Nis Wegmann[3]

[1] Galois, Inc. leepike@galois.com
[2] National Institute of Aerospace sebastian.niller@nianet.org
[3] University of Copenhagen wegmann@diku.dk

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, and we describe a case study in a Byzantine fault-tolerant airspeed sensor system.

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 [1]. 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 [2]. Nearly 20 years ago, Butler and Finelli showed that testing alone cannot verify the reliability of ultra-critical software [3].

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.

Contributions. Despite the relevance of RV to ultra-critical systems, there has been relatively little research on RV in that context. One of the primary contributions of this paper is to place RV within that context, particularly describing the constraints any RV solution must satisfy. A second contribution is the introduction of the notion of "easy fault-tolerance", where the machinery for implementing fault-tolerance resides in the monitor rather than the system under observation. Our third contribution is Copilot: a Haskell-based open-source language, compiler, and associated verification tools for generating RV monitors. Copilot answers two questions: (1) "Is RV possible for ultra-critical systems?" and (1) "Can functional programming be leveraged for embedded system RV?" We attempt to answer these questions by presenting the use of Copilot in a case-study replicating airspeed sensor failures in commercial aircraft.

Outline. We describe three recent software-related aircraft and Space Shuttle incidents motivating the need for RV in Section 2. In Section 3, we describe the constraints of RV implementations in the context of ultra-reliable systems.We describe the language Copilot in section 4; specifically, we describe how Copilot provides "easy" fault-tolerance, and we describe our approach to generating highly-reliable monitors. We present our use of Copilot in a case study simulating an ultra-reliable air speed system in Section 5. The remaining two sections present related work and conclusions, respectively.

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 [4] and the Ariane 5 Flight 501 disaster [5]. 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 [6]. 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 [7]), 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 [8]. 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) [9]. 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 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 [10]) 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) [11] and Monitor Oriented Programming (MOP) [12] 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 [13]. Other work at the University of Waterloo also investigates time-triggered RV [14, 15].

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).

4 Copilot: a Language for Ultra-Critical RV

To answer the challenge of RV in the context of fault-tolerant systems, we have developed a stream language called *Copilot*.[4] Copilot is designed to achieve the "FaCTS" constraints described in Section 3.

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

```
engineMonitor = do
  trigger "shutoff" (not overHeat) [arg maj]
  where
  vals     = map externW8 ["tmp_probe_0", "tmp_probe_1", "tmp_probe_2"]
  exceed   = map (< 250) vals
  maj      = majority exceed
  checkMaj = aMajority exceed maj
  overHeat = (extern "cooler" || (maj && checkMaj)) ‘since’ not maj
```

Fig. 1. A safety property and its corresponding Copilot monitor specification.

[4] Copilot is released under the BSD3 license and pointers to the compiler and libraries can be found at `http://leepike.github.com/Copilot/`.

While a preliminary description of the language has been presented [13], significant improvements to the language have been made and the compiler has been fully reimplemented. In any event, the focus of this paper is the unique properties of Copilot for implementing hardware fault-tolerance and software monitoring in the context of an ultra-critical system. Copilot is a language with stream semantics, similar to languages like Lustre [16]; we mention advantages of Copilot over Lustre in Section 6.

To briefly introduce Copilot, we provide an example Copilot specification in Figure 1. A Copilot monitor program is a sequence of *triggers*. A trigger is comprised of a name (`shutoff`), a Boolean guard (`not overHeat`), and a list of arguments (in this case, one argument, `maj`, is provided). If and only if the condition holds is the function `shutoff` called with the arguments. What a trigger does is implementation-dependent; if Copilot's C code generator is used, then a raw C function with the prototype

```
void shutoff(uint8_t maj);
```

should be defined. Within a single Copilot program, triggers are scheduled to fire synchronously, if they fire at all. Outside of triggers, a Copilot monitor is side-effect free with respect to non-Copilot state. Thus, triggers are used for other events, such as communication between monitors, as described in Section 4.2.

A trigger's guard and arguments are *stream* expressions. Streams are infinite lists of values. The syntax for defining streams is nearly identical to that of Haskell list expressions; for example, the following is a Copilot program defining the Fibonacci sequence.

```
fib = [0, 1] ++ fib + drop 1 fib
```

In Copilot streams, operators are automatically applied point-wise; for example, negation in the expression `not overHeat` is applied point-wise over the elements of the stream `overHeat`. In Figure 1, the streams are defined using library functions. The functions `majority`, `aMajority`, and `‘since’` are all Copilot library functions. The functions `majority` (which determines the majority element from a list, if one exists—e.g., `majority [1, 2, 1, 2, 1] == 1`) and `aMajority` (which determines if any majority element exists) come from a majority-vote library, described in more detail in Section 4.1. The function `‘since’` comes from a a past-time linear temporal logic library. Libraries also exist for defining clocks, linear temporal logic expressions, regular expressions, and simple statistical characterizations of streams.

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). We rely on the type system to ensure the Copilot compiler is type-correct. 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.

To sample values from the "external world", Copilot has a notion of *external variables*. External variables include any value that can be referenced by a C variable (as well as C functions with a non-void return type and arrays of values). In the example, three external variables are sampled: `tmp_probe_0`, `tmp_probe_1`, `tmp_probe_2`. External variables are lifted into Copilot streams by applying a typed "extern" function. For example, An expression `externW8 "var"` is a stream of values sampled from the variable `var`, which is assumed to be an unsigned 8-bit word.

Copilot is implemented as an *embedded domain-specific language* (eDSL). An eDSL is a domain-specific language in which the language is defined as a sub-language of a more expressive host language. Because the eDSL is embedded, there is no need to build custom compiler infrastructure for Copilot—the host language's parser, lexer, type system, etc. can all be reused. Indeed, Copilot is *deeply embedded*, i.e., implemented as data in the host language that can be manipulated by "observer programs" (in the host language) over the data, implementing interpreters, analyzers, pretty-printers, compilers, etc. Copilot's host language is the pure functional language Haskell [17]. In one sense, Copilot is an experiment to answer the question, "To what extent can functional languages be used for ultra-critical system monitoring?"

One advantage of the eDSL approach is that Haskell acts as a powerful macro language for Copilot. For example, in Figure 1, the expression

```
map externW8 ["tmp_probe_0", "tmp_probe_1", "tmp_probe_2"]
```

is a *Haskell* expression that maps the external stream operator `externW8` over a list of strings (variable names). We discuss macros in more detail in Section 4.1.

Additionally, by reusing Haskell's compiler infrastructure and type system, not only do we have stronger guarantees of correctness than we would by writing a new compiler from scratch, but we can keep the size of the compiler infrastructure that is unique to Copilot small and easily analyzable; the combined front-end and core of the Copilot compiler is just over two thousand lines of code. Our primary back-end generating C code is around three thousand lines of code.

Copilot is designed to integrate easily with multiple back-ends. Currently, two back-ends generate C code. The primary back-end uses the Atom eDSL [18] for code generation and scheduling. Using this back-end, Copilot compiles into constant-time and constant-space programs that are a small subset of C99. By *constant-time*, we mean C programs such that the number of statements executed is not dependent on control-flow[5] and by *constant-space*, we mean C programs with no dynamic memory allocation.

The generated C is suitable for compiling to embedded microprocessors: we have tested Copilot-generated specifications on the AVR (ATmega328 processor) and STM32 (ARM Cortex M3 processor) micro-controllers. Additionally, the compiler generates its own static periodic schedule, allowing it to run on bare

[5] We do not presume that a constant-time C program implies constant execution time (e.g., due to hardware-level effects like cache misses), but it simplifies execution-time analysis.

hardware (e.g., no operating system is needed). The language follows a sampling-based monitoring strategy in which variables or the return values of functions of an observed program are periodically sampled according to its schedule, and properties about the observations are computed.

4.1 Easy Fault-Tolerance

Fault-tolerance is hard to get right. The examples given in Section 2 can be viewed as fault-tolerance algorithms that failed; indeed, as noted by Rushby, fault-tolerance algorithms, ostensibly designed to prevent faults, are often the source of systematic faults themselves [19]! One goal of Copilot is to make fault-tolerance easy—easy for experts to specify algorithms without having to worry about low-level programming errors and easy for users of the functions to integrate the algorithms into their overall system. While Copilot cannot protect against a designer using a fault-tolerant algorithm with a weak fault-model, it increases the chances of getting fault-tolerance right as well as decoupling the design of fault-tolerance from the primary control system. Finally, it separates the concerns of implementing a fault-tolerance algorithm from implementing the algorithm as a functionally correct, memory-safe, real-time C program.

As noted, because Copilot is deeply embedded in Haskell, Haskell acts as a meta-language for manipulating Copilot programs. For example, the streams `maj`, `check`, and `overHeat` in Figure 1 are implemented by Haskell functions that generate Copilot programs.

To see this in more detail, consider the Boyer-Moore Majority-Vote Algorithm, the most efficient algorithm for computing a majority element from a set[6] [20]. The `majority` library function implements this algorithm as a Copilot macro as follows:

```
majority (x:xs) = majority' xs x (1 :: Stream Word32)
  where
  majority' []     candidate _   = candidate
  majority' (x:xs) candidate cnt =
    majority' xs (if cnt == 0 then x else candidate)
                 (if cnt == 0 || x == candidate then cnt+1 else cnt-1)
```

The macro specializes the algorithm for a fixed-size set of streams at compile-time to ensure a constant-time implementation, even though the algorithm's time-complexity is data-dependent. (Our library function ensures sharing is preserved to reduce the size of the generated expression.)

As an informal performance benchmark, for the `majority` algorithm voting over five streams of unsigned 64-bit words, we compare C code generated from Copilot and constant-time handwritten C. Each program is compiled using `gcc -O3`, with a `printf` statement piped to `/dev/null` (to ensure the function is not optimized away). The hand-written C code is approximately nine percent faster.

[6] Due to space limitations, we will not describe the algorithm here, but an illustration of the algorithm can be found at `http://www.cs.utexas.edu/~moore/best-ideas/mjrty/example.html`.

While the Boyer-Moore algorithm is not complicated, the advantages of the Copilot approach over C are (1) `majority` is a polymorphic library function that can be applied to arbitrary (Copilot-supported) data-types and sizes of voting sets; with (2) constant-time code, which is tedious to write, is generate automatically; (3) the Copilot verification and validation tools (described in Section 4.3) can be used.

4.2 Distributed Monitoring

Our case study presented in Section 5 implements distributed monitors. In a distributed monitor architecture, monitors are replicated, with specific parameters per process (e.g., process identifiers). The meta-programming techniques described in Section 4.1 can be used to generate distributed monitors by parameterizing programs over node-specific data, reducing a tedious task that is traditionally solved with makefiles and C macros to a few lines of Haskell.

Copilot remains agnostic as to how the communication between distinct processes occurs; the communication can be operating system supported (e.g., IPC) if the monitors are processes hosted by the same operating system, or they can be raw hardware communication mechanisms (e.g., a custom serial protocol and processor interrupts). If the monitors are on separate processors, the programmer needs to ensure either that the hardware is synchronized (e.g., by using a shared clock or by executing a clock synchronization protocol). Regardless of the method, triggers, described above, are also used to call C functions that implement the platform-specific protocol. Incoming values are obtained by sampling external variables (or functions or arrays).

4.3 Monitor Assurance

"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. Copilot contains a custom QuickCheck [21]-like test harness that generates random Copilot programs and tests the interpreted output against the compiler to ensure correspondence between the two. We have tested millions of randomly-generated programs between the compiler and interpreter.

We use the CBMC model checker [22] to verify C code generated by Copilot specifications. CBMC provides an independent check on the compiler. CBMC can prove that the C code is memory-safe, including proving there are no arithmetic underflows or overflows, no division by zero, no not-a-number floating-point values, no null-pointer dereferences, and no uninitialized local variables. Some of these potential violations are impossible for the Copilot compiler to generate (e.g., null-pointer dereferences), provided it is bug-free. Sometimes CBMC cannot prove that a C program is memory-safe, since it requires the program to be loop-free. The C code generated by Copilot implements a state machine that generates the next values of the stream equations (see [13] for details). CBMC can symbolically unroll the state machine a small fixed number of steps. A separate (so far informal) proof must be given that the program has been unrolled sufficiently to prove memory-safety.

5 Case Study

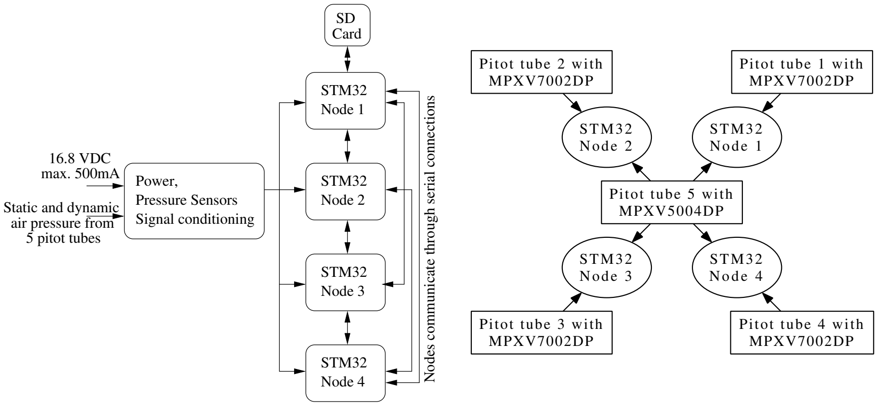

SD Card
STM32 Node 1
STM32 Node 2
STM32 Node 3
STM32 Node 4
16.8 VDC max. 500mA
Static and dynamic air pressure from 5 pitot tubes
Power, Pressure Sensors Signal conditioning
Nodes communicate through serial connections
Pitot tube 2 with MPXV7002DP
Pitot tube 1 with MPXV7002DP
Pitot tube 5 with MPXV5004DP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

Fig. 2. Hardware stack and pitot tube configuration.

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

We sample five pitot tubes, attached to the wings of an Edge 540 subscale aircraft. The pitot tubes provide total and static pressure that feed into one MPXV5004DP and four MPXV7002DP differential pressure sensors (Figure 2).

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 [7]). 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.

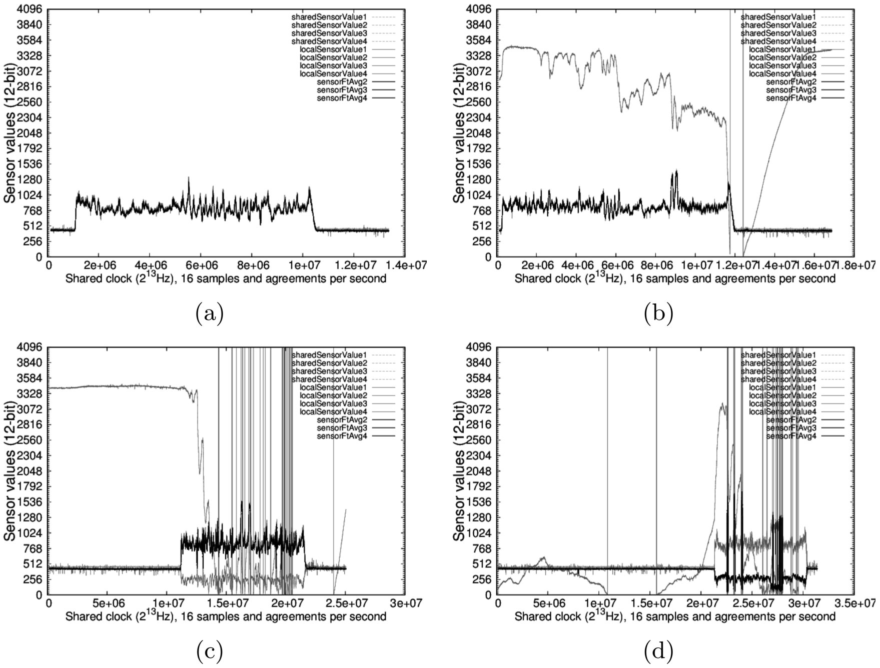
Monitors communicate over dedicated point-to-point bidirectional serial connections. With one bidirectional serial connection between each pair of nodes, the monitor bus and the processing nodes form a complete graph. All monitors on the nodes run in synchronous steps; the clock distribution is ensured 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 4.1) 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.



(a)

(b)

(c)

(d)

Fig. 3. Logged pressure sensor, voted and averaged data.

The graphs in Figure 3 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 3(a), no physical faults are introduced; in Figure 3(b), one benign fault has been injected by putting a cap over the total pressure probe of one local tube.[7] In Figure 3(c), in addition to the capped tube, sticky tape is placed over another tube, and in Figure 3(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 3(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 3(b), the cap on the capped tube creates a positive offset on the dynamic pressure as well as turbulences and low pressure on the static probes. At 1.2E7 clock ticks, the conversion and calibration function of the stuck tube results in an underflowing value. In Figure 3(d), with

[7] Tape left on the static pitot tube of Aeroperú Flight 603 in 1996 resulted in the death of 70 passengers and crew [24].

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.

Discussion. The purpose of the case-study is to test the feasibility of using Copilot-generated monitors in a realistic setting to "bolt on" fault-tolerance to a system that would otherwise be lacking that capability. The Copilot agreement monitor is around 200 lines. The generated real-time C code is nearly 4,000 lines.

Copilot reduced the effort to implement a non-trivial real-time distributed fault-tolerant voting scheme as compared to implementing it directly in C. While a sampling-based RV approach works for real-time systems, one major challenge encountered is ensuring the monitor's schedule corresponds with that of the rest of the system. Higher-level constructs facilitating timing analysis would be beneficial. Furthermore, it may be possible to reduce the size of the monitor's C code using more aggressive optimizations in the Copilot compiler.

6 Related Work

Using RV to implement fault-tolerance can be considered to be a "one-out-of-two" (1oo2) architecture [25], similar in spirit to the idea of the Simplex architecture [26]. In a 1oo2 architecture, one component may be an arbitrarily complex control system, and the other component is a monitor.

Copilot shares similarities with other RV systems that emphasize real-time or distributed systems. Krüger, Meisinger, and Menarini describe their work in synthesizing monitors for a automobile door-locking system [27]. 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 [28]. 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). [29].

There are few instances of RV focused on C code. One exception is Rmor, which generates constant-memory C monitors [30]. 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 time-triggered RV for arbitrary programs, particularly for testing [14, 15]. 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 [18], which we use as a back-end, Feldspar is an eDSL for digitial signal processing [31]. Copilot is similar in spirit to other languages with stream-based semantics, notably represented by the Lustre family of languages [16]. 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" [32], 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 Section 4.1, that get monomorphised at compile-time.

7 Conclusions

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.

Some research directions that remain include the following. Stochastic methods might be used to distinguish random hardware faults from systematic faults, as the strategy for responding to each differs [33]. 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. Because we have a sampling-based monitoring strategy, we would also like to be able to infer the periodic sampling rate required to monitor some property.

Research developments in RV have potential to improve the reliability of ultra-critical systems, and we hope a growing number of RV researchers address this application domain.

Acknowledgements

This work is supported by NASA Contract NNL08AD13T. We thank Ben Di Vito and Alwyn Goodloe at NASA Langley for their advice. Robin Morisset contributed to an earlier version of Copilot. NASA Langley's AirSTAR Rapid Evaluation and Prototyping Group graciously provided resources for our case study.

References

1. Rushby, J.: Software verification and system assurance. In: Intl. Conf. on Software Engineering and Formal Methods (SEFM), IEEE (November 2009) 3–10
2. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. Communications of the ACM (CACM) **53**(6) (June 2010) 107–115

3. Butler, R.W., Finelli, G.B.: The infeasibility of quantifying the reliability of life-critical real-time software. IEEE Transactions on Software Engineering **19** (January 1993) 3–12
4. Leveson, N.G., Turner, C.S.: An investigation of the Therac-25 accidents. Computer **26** (1993) 18–41
5. Nuseibeh, B.: Soapbox: Ariane 5: Who dunnit? IEEE Software **14**(3) (1997) 15–16
6. Bergin, C.: 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/`.
7. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems **4** (July 1982) 382–401
8. Australian Transport Safety Bureau: In-flight upset event 240Km North-West of Perth, WA Boeing Company 777-200, 9M-MRG 1 August 2005. ATSB Transport Safety Investigation Report (2007)
9. Macaulay, K.: ATSB preliminary factual report, in-flight upset, Qantas Airbus A330, 154 Km West of Learmonth, WA, 7 October 2008. Australian Transport Safety Bureau Media Release (November 14 2008) Available at `http://www.atsb.gov.au/newsroom/2008/release/2008_45.aspx`.
10. RTCA: Software considerations in airborne systems and equipment certification. RTCA, Inc. (1992) RCTA/DO-178B.
11. Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I., Sokolsky, O.: Formally specified monitoring of temporal properties. In: 11th Euromicro Conference on Real-Time Systems. (1999) 114–122
12. Chen, F., Roşu, G.: Java-MOP: a monitoring oriented programming environment for Java. In: 11th Intl. Conf. on Tools and Algorithms for the construction and analysis of systems (TACAS'05). Volume 3440 of LNCS., Springer (2005) 546–550
13. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: A hard real-time runtime monitor. In: Runtime Verification (RV). Volume 6418., Springer (2010) 345–359
14. Fischmeister, S., Ba, Y.: Sampling-based program execution monitoring. In: ACM International conference on Languages, compilers, and tools for embedded systems (LCTES). (2010) 133–142
15. Bonakdarpour, B., Navabpour, S., Fischmeister, S.: Sampling-based runtime verification. In: 17th Intl. Symposium on Formal Methods (FM). (2011)
16. Mikáĉ, J., Caspi, P.: Formal system development with Lustre: Framework and example. Technical Report TR-2005-11, Verimag Technical Report (2005) Available at `http://www-verimag.imag.fr/index.php?page=techrep-list&lang=en`.
17. Jones, S.P., ed.: Haskell 98 Language and Libraries: The Revised Report. http://haskell.org/ (2002)
18. Hawkins, T.: Controlling hybrid vehicles with Haskell. Presentation. *Commercial Users of Functional Programming* (CUFP) (2008) Available at `http://cufp.galois.com/2008/schedule.html`.
19. Rushby, J.: Formalism in safety cases. In Dale, C., Anderson, T., eds.: Making Systems Safer: Proceedings of the Eighteenth Safety-Critical Systems Symposium, Bristol, UK, Springer (February 2010) 3–17 Available at `http://www.csl.sri.com/users/rushby/papers/sss10.pdf`.
20. Boyer, R.S., Moore, J.S.: Mjrty: A fast majority vote algorithm. In: Automated Reasoning: Essays in Honor of Woody Bledsoe. (1991) 105–118
21. Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of haskell programs. In: ACM SIGPLAN Notices, ACM (2000) 268–279

22. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, Springer (2004) 168–176
23. 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`.
24. Ladkin, P.B.: News and comment on the Aeroperu b757 accident; AeroPeru Flight 603, 2 october 1996 (2002) Online article RVS-RR-96-16. Available at `http://www.rvs.uni-bielefeld.de/publications/Reports/aeroperu-news.html`.
25. Littlewood, B., Rushby, J.: Reasoning about the reliability of diverse two-channel systems in which one channel is "possibly perfect". Technical Report SRI-CSL-09-02, SRI (January 2010)
26. Sha, L.: Using simplicity to control complexity. IEEE Software (July/August 2001) 20–28
27. Krüger, I.H., Meisinger, M., Menarini, M.: Runtime verification of interactions: from MSCs to aspects. In: International Conference on Runtime Verification, Springer (2007) 63–74
28. Chen, F., d'Amorim, M., Roşu, G.: Checking and correcting behaviors of java programs at runtime with Java-MOP. Electronic Notes in Theoretical Computer Science **144** (May 2006) 3–20
29. Bonakdarpour, B., Kulkarni, S.S.: SYCRAFT: A tool for synthesizing distributed fault-tolerant programs. In: International Conference on Concurrency Theory (CONCUR '08), Springer (2008) 167–171
30. Havelund, K.: Runtime verification of C programs. In: Testing of Software and Communicating Systems (TestCom/FATES), Springer (2008) 7–22
31. Axelsson, E., Claessen, K., Dvai, G., Horvth, Z., Keijzer, K., Lyckegrd, B., Persson, A., Sheeran, M., Svenningsson, J., Vajda, A.: Feldspar: a domain specific language for digital signal processing algorithms. In: 8th ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign. (2010)
32. Halbwachs, N., Raymond, P.: Validation of synchronous reactive systems: from formal verification to automatic testing. In: ASIAN'99, Asian Computing Science Conference, LNCS 1742, Springer (December 1999)
33. Sammapun, U., Lee, I., Sokolsky, O.: RT-MaC: runtime monitoring and checking of quantitative and probabilistic properties. In: 11th IEEE Intl. Conf. on Embedded and Real-Time Computing Systems and Applications. (2005) 147–153