Model-based Real-time Testing of Drone Autopilots

Andrea Patelli* and Luca Mottola*+
*Politecnico di Milano (Italy), +SICS Swedish ICT
Contact author: luca.mottola@polimi.it

ABSTRACT

Key to the operation of robot drones is the autopilot software that realizes the low-level control. The correctness of autopilot implementations is currently mainly verified based on simulations. These may overlook the timing aspects of control loop executions, which are however fundamental to dependable operation. We report on our experience in applying model-based real-time testing to Ardupilot, a widely adopted autopilot. We describe our approach at deriving a model of Ardupilot's core functionality and at reducing the model to enable practical testing. Our work reveals that Ardupilot may fail in meeting the time constraints associated to critical functionality, such as enabling fail-safe operation. Through controlled experiments, we demonstrate the real-world occurrence of such erroneous executions.

1. INTRODUCTION

Aerial vehicles, ground robots, and aquatic rovers enable sophisticated applications [5, 16, 24]. Key to their dependable operation is the *autopilot* software that implements the low-level motion control. Autopilots process various sensor inputs, such as accelerations and GPS coordinates, to operate the electrical motors that set the vehicle's *attitude*. The high-level inputs come from a ground-control station (GCS) where mission parameters are configured, or from a human operator who drives the drone using a remote controller.

Problem. The autopilot is responsible for a number of safety functionality triggered in response to faults. For example, it may force the drone to land should the battery voltage drop below a threshold. Correctly implementing this functionality is fundamental: accidents are often reported whose causes may be attributed, in a way or the other, to malfunctioning autopilots [19].

Implementing autopilot software is, nevertheless, challenging. Size, cost, and energy concerns require autopilots to run on resource-constrained embedded hardware. This forces developers to employ low-level languages and to aggressively optimize implementations. Moreover, autopilot software operates in a setting where developers struggle to gain run-time information useful for verifying the correctness. Besides the hurdles at testing autopilots in real executions, resource-constraints hamper extensive logging and monitoring.

To complicate matters, many autopilot functionality need to execute in real-time, that is, subject to given time constraints. A paradigmatic example is that of safety functionality, which must be operated within strict time bounds to be effective. Should these bounds be violated, the drone may harm surrounding people or objects. Testing real-time software, however, is challenging even in mainstream computing, as testified by the bast literature on the subject [12].

The result of such a state of affairs is that autopilot software often undertakes little testing compared to the key role it plays. Further, most such testing is essentially performed in simulations and looks at functional requirements. However, simulations provide little coverage, whereas non-functional requirements such as real-time functionality are tackled by over-provisioning the hardware so that the software runs "fast enough" not to cause problems in everyday use. This provides no definitive guarantee on the correct functioning of autopilots, which is unacceptable especially for safety functionality.

Contribution. This paper presents our work and experience at gaining a deeper insight into the proportions, depth, and implications of the issues above.

We take Ardupilot [3], a widely employed open-source autopilot implementation we describe in Sec. 2, and apply state-of-the-art model-based real-time testing techniques for verifying the correctness of crucial safety functionality. As illustrated in Sec. 3, the techniques we adopt are both *exhaustive*, that is, given certain inputs, they can verify if developer-provided correctness properties hold in *every possible* execution, and operate *off-line*. The latter feature is immensely useful to overcome the aforementioned practical hurdles in testing autopilots right on the drones.

We build a model of Ardupilot's core functionality in two steps. Sec. 4 describes how we derive the code's control flow graph in a fully automated way, which guarantees the model is structurally equivalent to the code. Next, we perform extensive measurements of the execution times of Ardupilot's code over a Pixhawk board in a minimally invasive way, using custom hardware and during actual executions.

The model derived this way, however, turns out impractical to carry out the actual testing process. As it is common when applying exhaustive techniques, we stumble upon state space explosion problems that hamper the conclusion of the verification process [13, 25]. To overcome these, Sec. 5 illustrates model reduction techniques we apply to shave the

DroNet'16, June 26 2016, Singapore, Singapore
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4405-0/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2935620.2935630

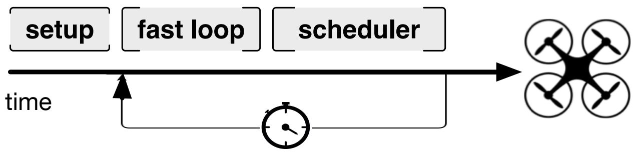

Figure 1: Ardupilot's control loop.

state space, ultimately obtaining a model usable on ordinary personal computers.

The outcome of the testing process on the reduced model, described in Sec. 6, indicates that, for example, specific executions *are possible* where Ardupilot's fail-safe mode might be *arbitrarily delayed*. This means that, even though a fault happens that requires the fail-safe mode to be enabled, the latter might not operate immediately. Rather, the drone continues working (or not working) as in the absence of the fault. Through real-world controlled experiments using a custom-built aerial drone, we demonstrate the not-so-rare occurrence of such erroneous executions.

We conclude the paper by offering in Sec. 7 an outlook on how the insights we present in this work may represent an input to further developments in the field.

2. BACKGROUND

We describe the autopilot we consider and cast our work in the broader context.

2.1 Ardupilot

Ardupilot is an open-source project [3] at the basis of many commercial products, including those of 3DRobotics [1] and many others, and boasts a large on-line community. It well exemplifies the state of the art in the field of autopilot development, platforms, and testing. Other existing autopilots, such as OpenPilot [17] and Cleanflight [10], show similar characteristics.

Implementation. Fig. 1 shows the execution of Ardupilot's control loop. The *fast loop* only includes critical motion control functionality. The time left from the execution of *fast loop* is given to an application-level *scheduler* that distributes it among non-critical tasks, such as logging. The scheduler operates in a *best-effort* manner based on programmer-provided priorities.

Initially, *fast loop* processes values from the Inertial Measurement Unit (IMU), which provides an indication of the vehicle's forces in the three dimensions. IMU information is combined with GPS readings to determine updated attitude control by minimizing the error between the desired and actual pitch, roll, and yaw. This information is then converted into commands sent to the motors to orient the vehicle.

Ardupilot runs on various embedded hardware. A primary example is the Pixhawk board [18], which features a Cortex M4 MCU at 168 MHz and a full sensor array for navigation. On Pixhawk boards, Ardupilot's control loop is statically configured to run at 400 Hz.

Testing. The correctness of Ardupilot's implementation is currently mainly verified using simulations. Ardupilot's distribution includes the Software In The Loop simulator (SITL) [3], which allows one to run Ardupilot on ordinary machines. SITL offers the same APIs as a real instantiation of Ardupilot on embedded hardware. The drone behavior is rendered based on a model of flight dynamics embedded within SITL. A GCS can connect to the simulated Ardupilot instead of a real drone, and issue commands to drive waypoint navigation and remotely log the mission execution.

Using SITL provides a practical means to quickly test functional properties of Ardupilot, or to check Ardupilot's behavior in specific circumstances. Further, tests can be scripted. This allows one to build automatic testing frameworks, for example, to perform regression testing on nightly builds as new features are added.

2.2 Related Work

In embedded software, the lack of exhaustiveness is widely acknowledged as a drawback of simulation-based testing [8]. The testing outcome only applies to that specific execution with given inputs. Simulations may also fall short in realism [8], for example, as it is difficult to accurately model the execution times of embedded hardware. However, testing on top of resource-constrained hardware is likely difficult without generating "heisenbugs": fictitious software defects induced by probing the executions [15].

Model-based testing complements simulations [11] by operating on an abstract representation of the system, which allows to test a system without concretely running it. The models are input to a tool together with developer-provided *properties*, that is, an encoding of what is considered the correct behavior. The tool returns whether the model satisfies the properties in *every possible execution*, and is therefore exhaustive. If a property is violated, a counter-example is produced that shows an execution where a property does not hold. Because of these features, model-based testing is widely used in safety critical software [8].

Applying model-based testing to the special case of real-time software requires dedicated models and tools [12]. For example, various kinds of timed automata [11, 13] to model the behavior of the system exist. Different model representations require different algorithms to check the properties of interest [13, 25]. A vast literature exists on the subject [12], which shows how these techniques can be effectively applied to a number of different application domains.

Among these domains is also the one of avionics [6, 21, 22]. However, compared to traditional avionic systems, robot drones are peculiar in at least three respects. First, many state-of-the-art systems, including Ardupilot, are the result of a community-driven unstructured development process that favors agile methods and tools [14] over more systematic ones, such as model-based testing. Second, the hardware to run avionics software is often carefully dimensioned and custom-zed to the specific functionality; resource constraints rarely represent an issue [21]. In robot drones, however, off-the-shelf hardware is employed to keep costs down. Finally, unlike traditional avionic systems, the operation of robot drones depend on a long-range wireless connection to the GCS or the user's remote controller, whose failure drastically impact the device's operation.

3. ARDUPILOT MODEL-BASED TESTING

We apply model-based testing to Ardupilot's core functionality, that is, the *fast loop* portion of Fig. 1, and particularly focus on Ardupilot's *failsafe* flight mode. The latter should be automatically enabled, for example, whenever no valid signal is received from the telemetry radio or the remote controller for a given time period. We next describe the properties and tools we consider. Sec. 4 illustrates how we build an Uppaal model to represent Ardupilot's processing relevant to check these properties.

Properties. We examine three classes of properties. We test two *reachability* properties [2], that is, properties specifying that certain critical portions of the code should be reachable in every possible execution:

R1: the function responsible for processing the signals from the telemetry radio or the remote controller, should terminate for every possible input;
R2: it should be possible to enable the failsafe flight mode should the need arise as described above.

We also consider three *safety* properties [2] stating that unwanted situations should never happen:

S1: the worst-case execution time of *fast loop* should be at most T_w, not to impact the resulting rate of control loop executions;
S2: there should be no executions of *fast loop* that result in a deadlock that indefinitely prevents enabling the failsafe mode;
S3: the time that *fast loop* takes to enable the failsafe mode should never exceed T_f.

In Ardupilot, the actual values for T_w and T_f are platform-dependent. Finally, we also examine one *liveness* property [2] that indicates a desirable condition that should eventually occur:

L1: after the failsafe mode is enabled, re-gaining a signal from the telemetry radio or the remote controller eventually disables it.

Tools. We use Uppaal [13], a state-of-the-art environment for simulation and testing of real-time systems. Uppaal is appropriate for systems that can be modeled as a collection of non-deterministic processes with real-valued clocks, communicating via channels or shared variables. Uppaal's target applications include closed-loop controllers and communication protocols.

Uppaal requires the input models to be specified as a network of timed automata. Sec. 4 describes how we build such a model for Autopilot's core functionality. Properties in Uppaal are specified using a subset of Timed Computation Tree Logic (TCTL) [2]. The translation of the properties above in TCTL is straightforward; the most complex being **L1** that requires using three temporal operators.

4. MODEL CONSTRUCTION

We build the model for Uppaal in two steps. First, we derive a state machine from the code's control flow graph. Next, we add execution times for each state by profiling real executions of Ardupilot.

4.1 State Machine

We use Scitool's Understand [20] to automatically derive the control flow graph for *fast loop* in Ardupilot 3.1.0. Understand is a static analysis tool often employed to reverse-engineer safety critical software.

Fig. 2 shows a snippet of Understand's output for the function processing the inputs from the remote controller. Using Understand, we recursively generate the control flow graph for every function possibly executed within *fast loop*, along with the connections between the individual control flow graphs due to function calls. We implement a post-processor that automatically transforms an XML-based representation of Understand's output into Uppaal's input format.

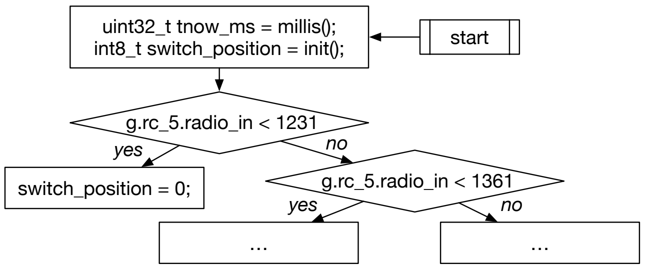

Figure 2: Snippet of the control flow graph for Ardupilot's function that processes inputs from the remote controller.

Such a processing, however, does not suffice to represent the behavior of *fast loop*:

1. Understand does not represent function calls due to interrupt handlers, as they are not visible in the source code. To cater for these executions, every time the code runs with interrupts enabled, our post-processor adds non-deterministic transitions in the Uppaal model that lead to the control flow of interrupt handlers.
2. Understand cannot render the semantics of timers in Ardupilot's code. To this end, we create an Uppaal sub-model that counts the number of time units accumulated in the main model. Whenever the counting reaches a threshold, a shared Boolean variable is flipped to signal the timer expired. This causes the model execution to enter the control flow of the function attached to the timer.

The resulting Uppaal model for Ardupilot's *fast loop* includes about 10,000 states connected by about 90,000 transitions. Note that a single state in the model does not necessarily map to a single instruction in the code. Whenever possible, Understand groups instructions in the same state, as shown at the top of Fig. 2. This does not entail that those instructions are executed atomically; the non-deterministic transitions we add ensure that interrupt handlers may preempt the model execution at any point in time, including between instructions that Understand groups together.

4.2 Execution Times

The model above still lacks information on the execution times in every state. Multiple approaches are possible to gain this information. One may estimate the number of machine instructions corresponding to every C++ instruction in Ardupilot, based on the target MCU architecture. An alternative would be to run the code in a suitable emulator. Finally, one may attempt to profile real executions.

Counting the machine instructions is likely to be inaccurate, as it cannot account for features such as instruction pipelines available on Cortex M4 MCUs. Emulators for Cortex M MCUs exist; however, they would need to be heavily customized to account for the variable times in interacting with the specific sensor hardware on Pixhawk boards. We therefore opt to profile real executions, which would return precise results only by ensuring that we only minimally alter the executions.

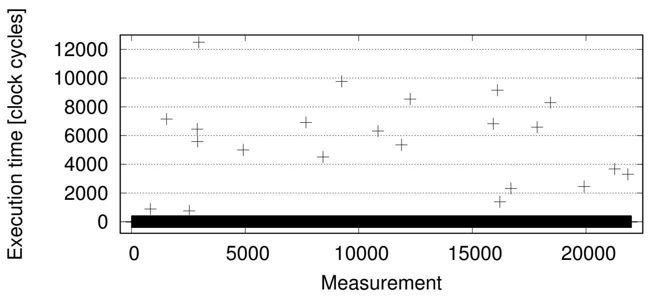

Figure 3: Results from profiling the execution time at *one sample state.* *Some measurements show very large values due to the preemption of the main code by interrupt handlers.*

Profiling. We connect a separately-powered RaspberryPI board to the GPIO pins of a Pixhawk board, and instrument Ardupilot's code to signal through the GPIO pins the transitions between the states of the Uppaal model. The RaspberryPI timestamps all such transitions and logs them asynchronously on local storage.

We already verified that such an approach is minimally invasive [7]: the added processing times in Ardupilot amount to a few MCU cycles and, most important, are *deterministic*. We can therefore subtract these times when post-processing the logs. Other than this, the added weight of the RaspberryPI and of its small battery, which only needs to power the board for the 20 minute duration of a flight, are negligible.

We total more 30 hours of flight using a custom-built quadcopter, a custom-built hexacopter, and a 3DR Y6 drone in both indoor and outdoor settings, alternating the use of a telemetry radio and a remote controller. To make sure we obtain the needed time information for every state in the Uppaal model, we manually force all executions of interest; for example, by artificially creating the conditions for Ardupilot to enter the failsafe mode because of a failure of the telemetry radio.

Results. The measurements turn out deceptive, as they show more variability than expected.

Fig. 3 exemplifies this aspect by showing the measurements during a single flight at *one* sample state that includes three assignments of 32 bit integer variables. Most of the measurements report very small values, as expected. A small fraction of the measurements, however, indicates that the MCU takes thousands of clock cycles to execute the three instructions.

Further investigations reveal that the variability in the measurements is due to the way hardware interrupts are managed. To ensure that interrupts are immediately served, especially those triggered by navigation sensors when a new reading is available, most of Ardupilot's processing in *fast loop* is preemptable. This means that a function's execution may be suspended for an arbitrary amount of time, until all pending interrupts are served. Such a design choice trades low latency in serving interrupts for deterministic execution times, which may however impact the time dynamics of control loop executions.

5. MODEL REDUCTION

The model we obtain is sufficient to check the properties of Sec. 3 using Uppaal, yet it is not practical. To be exhaustive, the checking process needs to explore all possible executions of the code to verify whether a certain property holds. Thus, Uppaal explores all model executions that correspond to different assignments of variables in the code. This creates huge processing demands. With the model in Sec. 4, checking the reachability properties fails because Uppaal reaches the memory allocation limits for single processes on our Linux machine. Differently, we give up on checking the liveness property after letting Uppaal run for two days on an Intel Xeon E3 processor clocked at 3.2 GHz.

We apply several reduction techniques to make the model practical. Some of these are domain-specific:

Radio channel abstraction: in Ardupilot, the execution is often driven by comparisons of the radio or sensor inputs against pre-defined thresholds. One example is in Fig. 2, where the code executes differently by comparing the input from radio channel 5 with two fixed values. In these cases, the actual input value is irrelevant as long as we can distinguish the cases leading to different execution paths. Thus, whenever possible in the Uppaal model, we reduce the domain of such decision variables to be of a cardinality equal to the number of execution paths they may determine. This limits the number of different executions Uppaal needs to explore.

Flight mode abstraction: depending on the flight mode, the admitted ranges of radio and sensor inputs change. For example, when the *super simple* mode is enabled, the admissible ranges of inputs from the remote controller is further limited to facilitate novice pilots. Ardupilot includes a sizeable amount of code merely to check these ranges and react accordingly. Similar to the case above, the concrete values of the inputs are immaterial as long as we can discern whether they are within or outside the admitted range. We thus restrict the domains of all relevant variables as done above, further limiting the number of different executions.

We also apply a number of domain-agnostic model reduction techniques. One example is the reset of variables [13]. If a variable **v** is never used again after state s, its value becomes immaterial. For all states that follow s in an execution, however, variable **v** is still part of the program's data and concurs to make executions different. By resetting **v** right after s to a fixed value, we help Uppaal join executions that would be different only for the value of a variable that is however immaterial.

The reduced model makes it possible to verify the properties of Sec. 3 in a matter of minutes on an ordinary PC. We discuss next the outcome of the process.

6. TESTING OUTCOME

Key to the testing process is how to account for the execution times we described in Sec. 4.2. We consider five ways to represent their variability:

Average assigns a state with the average execution time out of all measures we gather.

Min-max asks Uppaal to check all time values within an interval from the minimum to the maximum execution time we record.

Extreme considers the min-max interval on a post-processed set of measures obtained by applying the "extreme" outlier filtering [23].

Property	Min-max	Average	Extreme	Mild	Percentile
R1	Yes	Yes	Yes	Yes	Yes
R2	Yes	Yes	Yes	Yes	Yes
S1	**No**	Yes	Yes	**No**	Yes
S2	Yes	Yes	Yes	Yes	Yes
S3	**No**	Yes	**No**	**No**	Yes
L1	**No**	Yes	Yes	**No**	Yes

Figure 4: Outcome of testing process depending on variable execution times. *Time-sensitive properties are verified as a function of the impact on interrupt handlers.*

Mild considers the min-max interval on a post-processed set of measures obtained by applying the "mild" outlier filtering [23].

Percentile considers the min-max interval on a post-processed set of measures obtained by only considering the 99% percentile.

Results. Depending on the above, some properties may or may not be verified, as shown in Fig. 4.

On the bright side, the reachability properties are constantly verified, as expected given these are not time sensitive. Moreover, property **S2**, which checks the absence of deadlocks, is also verified no matter how the variable execution times are factored in. This gives us confidence in the quality of the code.

The remaining properties consistently fail when the execution times cover the **Min-max** interval. This entails taking into account all possible circumstances, no matter how rare they are: even if we recorded a single excessively long execution time in a state, that would be considered as the maximum value of the interval. As we exemplified in Fig. 3, there may be cases where such a maximum is orders of magnitudes larger than the most common values.

As for property **S1**, the worst-case execution time for *fast loop* on the Pixhawk should be $T_w = 2.5$ ms, as the loop is configured to run at 400 Hz. This already represents an undesirable situation, in that *fast loop* would eat the entire processing time for a single iteration, leaving no time for the *scheduler* part of the loop. Our analysis indicates that T_w is easily exceeded if interrupt handlers preempt the execution, as discussed in Sec. 4.2. This means that control slows down, and the drone looses reactivity.

The outcomes related to the management of the failsafe mode are possibly more notable. As for property **S3**, on the Pixhawk the upper bound to enable the failsafe mode is set to $T_f = 500$ ms (2000 ms) for failures of the remote controller (telemetry radio). Uppaal returns that these bounds are plainly violated as a result of interrupt handlers preempting the execution of *fast loop*.

As for property **L1**, Uppaal indicates that if the failsafe mode is enabled late, it may never be disabled even though the signal from the telemetry radio or the remote controller is re-acquired. The corresponding counter-example shows an inconsistency in the values of relevant variables: the failsafe is, in fact, operational, but some variables still refer to the previous flight mode.

The results of testing properties **S1**, **S3**, and **L1** using **Mild** outlier filtering are the same as in the **Min-max** case. Even though the counter-examples returned by Uppaal differ, this kind of filtering still does not suppress enough large values to change the outcome of the verification. When applying **Extreme** outlier filter, instead, only property **S3** keeps failing, again with a different counter-example.

Viceversa, properties **S1**, **S3**, and **L1** are verified when considering either the **Average** execution times or the 99% **Percentile**. Because the large values we record are somewhat "rare", they bear little impact on the average and likely fall outside the 99% percentile.

Back to reality. The results we obtain indicate that interrupt handlers are crucial. The question that naturally arises is then: *how rare is "rare"?*

To find an answer, we modify the equipment we described in Sec. 4.2 to monitor real executions searching for any of the counter-examples that Uppaal returns. Note that we are able to do so because we can leverage detailed information on the specific executions that violate a property; in other words, we know what we are looking for. Otherwise, monitoring real executions searching for any possible property violation would be prohibitively complex. By the same token, as Uppaal only returns the first counter-example found, it also means that a property may fail in a different way during a real execution, but that situation will go unnoticed.

We perform an additional 10 hours of flight using the same drones and settings of Sec. 4.2. To trigger the failsafe mode, we artificially disconnect the telemetry radio every 2 minutes, and reconnect it 10 seconds later. Throughout these experiments, we detect *twice* on two different drones the counter-example when property **S1** fails under the **Min-max** setting. Moreover, we detect *three times* on the same drone the counter-example when **S3** fails under the **Min-max** setting, and we recognize *once* the counter-example when property **S3** fails under the **Mild** setting.

7. DISCUSSION AND OUTLOOK

Notwithstanding the tremendous effort of Ardupilot's community, the efficient operation of the current implementation, and the fact that the telemetry radio is not expected to fail that often, a total of six property violations in 10 hours of operation are arguably significant for a potentially harmful system [19]. While the developer community of Ardupilot has confirmed the bugs we identified, some key developers also pointed out that no easy fix exists to these issues, and a significant re-design would be needed instead [4].

In a way, this is a reflection of the development process adopted for Ardupilot. The Ardupilot codebase changes often due to bug fixing or the introduction of new features, and with little overall coordination. In such a fluid community-driven setting, it is difficult to apply techniques such as model-based testing, in that someone should volunteer the effort to keep the models updated as the codebase changes.

On the other hand, we auspicate that autopilot software would follow the practice of larger open-source projects, such as the Linux kernel, where systematic testing techniques progressively integrated with agile development processes [9]. To this end, greater efforts are required from academia to customize techniques such as model-based testing for use in autopilot software.

Acknowledgments. The work was partly supported by the Projects "Zero-energy Buildings in Smart Urban Districts"

(EEB), "ICT Solutions to Support Logistics and Transport Processes" (ITS), and "Smart Living Technologies" (SHELL) of the Italian Ministry for University and Research.

8. REFERENCES

[1] 3D Robotics. UAV Technology. goo.gl/sBoH6.

[2] R. Alur et al. Model-checking for real-time systems. In *IEEE Symposium on Logic in Computer Science*, 1990.

[3] Ardupilot. Home. goo.gl/x2CHyM.

[4] Ardupilot Discussion Forums. Topic: "real-time violations", started February 11th, 2016. goo.gl/tIXfHH.

[5] BBC News. Disaster drones: How robot teams can help in a crisis. goo.gl/6efliV.

[6] T. Bienmüller et al. Formal verification of an avionics application using abstraction and symbolic model checking. In *Towards System Safety*. Springer, 1999.

[7] E. Bregu, D. Cantoni, N. Casamassima, L. Mottola, and K. Whitehouse. Reactive control of autonomous drones. In *MOBISYS*, 2016.

[8] B. Broekman and E. Notenboom. *Testing embedded software*. Pearson Education, 2003.

[9] A. Cimatti et al. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*. Springer, 2002.

[10] Cleanflight. Home. goo.gl/uCGmr4.

[11] A. En-Nouaary et al. Timed Wp-method: Testing real-time systems. *Software Engineering, IEEE Transactions on*, 28(11), 2002.

[12] R. L. Glass. Real-time: The "lost world" of software debugging and testing. *Comm. of the ACM*, 23(5), 1980.

[13] A. Hessel et al. Testing real-time systems using UPPAAL. In *Formal methods and testing*. Springer, 2008.

[14] R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[15] M. Musuvathi et al. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[16] F. Nex and F. Remondino. UAV for 3D mapping applications: A review. *Applied Geomatics*, 2003.

[17] OpenPilot. Home. goo.gl/D89lkb.

[18] PixHawk.org. PX4 autopilot. goo.gl/wU4fmk.

[19] Scientific American. 5 epic drone flying failures—and what the FAA is doing to prevent future mishaps. goo.gl/tIXfHH.

[20] SciTools. Code analysis with Understand. goo.gl/0VZGAc.

[21] J. Souyris et al. Formal verification of avionics software products. In *FM 2009:Formal Methods*. Springer.

[22] S. Thesing et al. An abstract interpretation-based timing validation of hard real-time avionics software. In *DSN*, 2003.

[23] J. Tukey. *Exploratory data analysis*. Reading, Mass., 1977.

[24] P. Wardle. Personal submersible drone for aquatic exploration. goo.gl/XTVgQF, 2015. US Patent App. 14/143,713.

[25] W. Yi et al. Automatic verification of real-time communicating systems by constraint-solving. In *Formal Description Techniques VII*. Springer, 1995.