How to Avoid Mistakes in Software Development for Unmanned Vehicles

Cezary Szczepański[a], Marcin Ciopcia[b]

The purpose of this paper is to propose a design and development methodology in terms of robustness of unmanned vehicle (UV) software development, which minimizes the risk of software failure in both experimental and final solutions. The most common dangers in UV software development were determined, classified, and analyzed based on literature studies and the authors' own experience in software development and analysis of open-source code. As a conclusion, "good practices" and failure countermeasures are proposed.

KEY WORDS
~ Unmanned vehicle
~ UV
~ UAV
~ Software development
~ Software robustness

a. Center of Space Technology, Institute of Aviation, Warsaw, Poland
e-mail: cezary.szczepanski@ilot.edu.pl
b. Mechanical Power Faculty, Wroclaw University of Science and Technology, Wroclaw, Poland
e-mail: marcin.ciopcia@pwr.edu.pl

doi: 10.7225/toms.v08.n02.005

1. PROBLEM IDENTIFICATION

Software for application in the unmanned vehicles (UV) is usually developed not by professional Functional Safety (FuSa) developers, but rather by people coming from other technical backgrounds related to the currently developed unmanned vehicles. One of the authors of this paper represents such an example, and the other one has a formal educational background in software development. The cases described here and their analyses allowed the authors to formulate some conclusions and directions on how to avoid mistakes in software development for unmanned vehicles' initial tests and how to make their further development smoother. This paper will not cover FuSa certification process guidance, such as a formal Hazard and Risk Analysis (HARA), but rather contains good practices for persons who e.g. want to test their new sensing or control algorithm safely. The discovered roots of failure can in most cases be avoided by using simple workflow rules and good programming practices. They can be defined and then easily incorporated by developers to increase robustness of the software and to eliminate or at least to minimize the chance of the developed vehicle's failure. The design rules presented here increase operational safety in the process of UV prototyping. Sharing them with less experienced programmers will have a positive impact on the quality of the final software. The awareness of risks related to UV software may also guide senior developers/software architects to apply the presented countermeasures in their projects, even at the expense of increased costs, time, software licensing, and similar.

The authors utilized their expertise in aeronautics in general, and in unmanned aerial vehicles' (UAV) software development in particular. Many of UAV software developers originate from open-

source projects. Applying the rules described below into these projects will have a significant impact on the hobbyists' work, who will benefit from increased reliability of their constructions. It may also be profitable for researchers, in whose case a smooth integration of experimental modules will simplify the prototype development process and increase the scientific progress of knowledge on UVs. This paper transfers "good practices" in software development into a compendium, which will help to increase software quality in UV research.

The terms of software development for UAV applications can be easily compared to the respective rules for manned aircraft. The main factor to be taken into consideration during the development of man piloted aviation software is the safety of the aircraft operation. For a UAV, one can formulate that the main factor is either a quick and cheap software development if the standard, open-source software cannot be applied, or free software available for that purpose to be adopted to the specific requirements of the developed project of a new or modernized UAV.

Software development on micro- and mini-class UAVs, which are the most popular among the users being also software developers, is not restricted by any standard. In contradiction to software for full-size aircraft, developers are not limited by certification requirements and standards, such as DO-178. From the researchers' point of view, this situation is highly beneficial as the whole development process can be drastically shortened. The lack of significant limitations enables them to use state-of-the-art solutions in terms of languages, coding techniques, software development tools, etc. The development of a new human-crewed aircraft, even using modern tools and processes, often takes more than ten years – mostly due to extensive certification and development procedure requirements. In terms of modern software development during such an extended period, almost all modern solutions often became obsolete.

Let us consider part of the newest standard (relative to 20/04/2019) for aviation software development, DO-332, which is a supplement to DO-178C and DO-278A. All these documents were introduced in 2011. DO-332 provides evaluation and acceptance criteria for Object-Oriented Programming (OOP) dedicated to aviation. The idea of OOP began to develop in the 1960s. The current "golden standard" for OOP embedded programming – C++ language – has been introduced in 1983, and nowadays compilers even support it for microcontrollers, such as ARM Cortex. Since 2011, three major standard revisions, C++11(ISO/IEC 14882:2011), C++14(ISO/IEC 14882:2014) and C++17(ISO/IEC 14882:2017) were published, but still will soon be replaced by already announced C++20.

As long as standards such as DO-178C do not forbid modern software development techniques such as Test-Driven Development (TDD) to be used alongside the formal, certified ones, the accepted techniques and tools may still not cover many useful features and capabilities of modern languages. This situation makes mentioned small UAVs to be excellent prototyping platforms for aviation software of the future.

Unfortunately, such lack of formal guidance in aviation may cause major hazards. All the procedures related to aviation software development are focused on safety. A small UAV may not cause such danger as an airliner. Nonetheless, the failure of onboard systems may still cause severe damages not only to the platform itself but also to people and objects around it. That is becoming even more important as the UAVs are being widely used not only by amateurs, but also by professional operators for many purposes, e.g. power grid screening, precision agriculture, picture and footage of private and mass events, cargo deliveries and many others. All the expanding areas of UAV applications are either visual line of sight (VLOS) or beyond visual line of sight (BVLOS) operations. Particularly the last mentioned type of operation could be seriously impacted by immature and not entirely safe software.

In the authors' professional career we heard statements such as "We crashed X prototypes – so our software is in an advanced state", or "Searching on an analytical solution, e.g. PID, is the domain of scientists and not "real engineers" who make money on their projects." In order to understand what is inherently wrong in such an approach, let us imagine that such words came out of your car tire design engineer. Would you entrust the life of your family in such a design attitude during a highway vacation trip?

The main goal of this article is to show that even for prototypes there is room for safety and robustness. Even when full, formal HARA is not required, it is worth remembering safety and some basic precautions leading to achieving it. We summarize below with explanations and comments what in our opinion are the most important ones. However, let us first remain with the end to which simple bugs might lead.

2. BRIEF LESSONS FROM THE HISTORY

In aviation software development, it is always worth remembering that even the smallest bug may cause a serious hazard. Those situations even happened in the most rigorous software development environments such as space missions or healthcare devices. The following list will briefly introduce a few examples which are considered the most tragic software failures in the history of software development.

(1) Atomic Energy of Canada Limited Therac-25

One of the most tragic software failures happened in Therac-25, produced by Atomic Energy of Canada Limited and used in medical examining. Those x-ray machines caused the death of at least six patients due to beta-radiation lethal overdose. The device had two modes of operation: weak electron beam and

x-rays scanning. The latter utilized a high-energy electron beam which was converted to the x-ray radiation within safety limits by a collimator and an ion chamber. Due to a faulty retraction of a conversion module in an x-ray modem caused by the software, patients were exposed to a high level of beta radiation, which caused severe damage to their bodies.

From the software side, this failure was caused by many factors and flaws in the design and implementation procedures. Reports from the investigation (Leveson et. al., 1993) show that many development process flaws, such as controlling a module position in an open-loop configuration (without a position detection), race conditions in software during fast keyboard typing in the control panel, a flag incrementation and lack of a proper review process during software development were present.

Conclusion: Poor software quality and pursuit of deadlines instead of quality might lead to fatal injuries.

(2) Mars Climate Orbiter

One of the critical sections of the software is its API. Simple errors such as unit mismatching may lead to tragic events. Such a case occurred in NASA's Mars Climate Orbiter project (Mars Climate Orbiter Mission Homepage, 2000). In the main specification of the project, the SI unit system was defined as the unit standard. Unfortunately, the part of the Martian ground approaching software delivered by an external contractor Lockheed Martin interpreted the specific impulse value as a $\frac{lbf}{s}$ instead of the specified $\frac{N}{s}$, which caused an error by factor ~4.5. The problem had not been detected before the mission launch and manifested itself during the Mars orbit insertion maneuver. A guidance system was designed to lead the spacecraft into the orbit 160 km above the Mars ground level in order to perform aerobraking. During this maneuver, the vehicle wrongly descended to 57 km, where atmospheric friction caused its overheating and destruction. The flight controllers in the Mission Control Centre spotted the deviation of the orbit and proposed to perform a Trajectory Correction Maneuver No. 5, but in the end the correction was not applied.

Conclusion: Mars Climate Orbiter mission, whose costs were estimated to $328 million, failed due to a simple failure to correct unit conversion error.

(3) Ariane 5

Another conversion error caused the failure of Flight 501, performed by Ariane 5 launcher (Lions, 1996). It led to the triggering of self-destruction sequence ~40 s after launch. The problem occurred in a layer of integration between the new and reused software already tested in filed subsystems coming from Ariane 4. One of the critical navigational values was calculated as a 64-bit floating-point and then converted to a 16-bit integer due to compatibility requirements. Unfortunately, the new, more powerful launcher design caused the value to overflow, which triggered operand error. That small error combined with unfortunate circumstances caused the catastrophic failure whose cost was estimated to $7 billion.

Conclusion: Lack of integration testing on the "proven in use" component cost $7 billion.

(4) the author's experience

Such errors may also occur for small UAVs. They are less expensive, but still may cause harmful situations for the operators and the environment. One of them happened to one of the authors during the research of the AHRS system for one of the projects. In order to implement a new AHRS subsystem, he had to prepare a new multirotor for flight. One of the steps of the setup procedure was a pre-flight PID tuning on a harness. The Ground Station (GS) software had managed all the regulators. Unfortunately, one of the programmers delivered hardcoded mode switch for experimental autonomous navigation system into the central repository, not passing the information on it to the other researchers. This small piece of code contained overwriting of the main flight mode settings and caused misconfiguration of PID controllers on the tuned vehicle. During a routine tune procedure, the vehicle fell into uncontrolled oscillations, which shortly led to an unexpected full-throttle command for one of the engines. A brushless motor, with 1,000 KV RPM constant at full speed combined with a 4s Li-Po battery and the 10-inch propeller may cause severe injuries. Fortunately, in our case it left only a code quality remainder in the form of a scar on one of the author's hands.

Conclusion: A temporary solution and lack of proper communication almost caused the author to lose his precious fingers.

3. GOOD PRACTICES IN SOFTWARE DEVELOPMENT

Remembering the above described cases and taking into consideration the UV software development conditions and priorities, the authors tried to formulate in a possibly clear way some advice for non-professional software developers. Also, professionals can apply some suggestions for their projects' performance.

(1) Write a developer-friendly code

Rule No. 1: Do not write software in a crypto style. Make it clear for any developer, even for yourself.

A valid code means, in principle, the code understandable to a machine. In order to make software development efficient, it also has to be understandable to a human. Modern compilers are very efficient in the optimization of assembly code. They are often considered as even superior to an experienced assembly programmer in terms of code performance.

Let us introduce an example: many language features, which were designed to enable low-level programmer manipulation, are currently obsolete. The keyword "register", which suggested old C compilers to keep a variable in the CPU register for easy and fast processing, is ignored by most modern compilers. Even authors of programming books do not include it in their new C language tutorials. Python developers also decided not to implement incrementing and decrementing operators, which were iconic for previous generations of programming languages. Iterable objects and language-specific inline loop operations replaced them.

Modern programmers are not as restricted in terms of writing machine-optimized code as they were a few years ago. They are permitted to write better quality, self-documenting code, which is oriented to ease of workout instead of performance optimization.

There comes one advice which came from the authors' experience: write your software in a way that even a 9-year-old child with the basic knowledge of programming could understand. If you have a part of a code which requires much effort to understand and to make it work – write a comment about the details.

Additional lines of information will not affect the code size nor performance, but will significantly save investigation effort of other programmers or even of the original author when some changes are required. A few additional minutes spent on writing them is a small fee to pay. Moreover, to write a sufficient comment, you have to understand the root cause of the problem and express it, which may protect you from accidental programming. It will also focus other programmers' attention on a problem – during e.g., reviewing process – which may trigger a discussion about refactoring lower levels of the software in order to avoid fighting the same problems over and over again.

It is also advisable not to use uncommon and sophisticated language constructs, especially if they do not bring any added value in terms of code size, performance, or improved readability. Usually, there is a good reason why even experts in a particular language do not use it. Often, the amount of time that would have to be consumed to understand and verify the correctness by the programmer who will work on this code in the future is unacceptable, and as a result, such sophisticated coding does not bring any added value to the project.

It is also beneficial not to forget about the underlying programming rules, such as a descriptive naming or separating code blocks by inline functions and comments with information about optimizations, tested solutions, and design decisions. The code written according to these rules will benefit from fewer bugs, due to simplicity. It leads us directly to the second rule of this guide.

(2) KISS

Rule No. 2: Do not complicate software in its structure or functionalities. Simple is beautiful and practical.

Acronym KISS is usually expanded as "Keep It Simple, Silly." It is often combined with another rule: "You Aren't Gonna Need It". Programming languages usually offer many ways to solve a particular problem. The above rule tells that the most straightforward way is almost always the best one. An overcomplicated structure and unnecessarily complex mechanisms not only makes software development harder, but also creates more opportunities for making a mistake. It is recommended to use complicated structures only when it is necessary in order to fulfil the requirements. Over-engineering often leads to performance drop, an increase in maintenance effort, and overall bug quantity in a code.

Usually, it is also beneficial not to implement functionalities in advance. It may cause an unnecessary increase in software's size, and it often leads to dead and untested code. If such an additional feature is required, a more intuitive and efficient way to implement it may be applied.

There is a good example of the authors' experience, which illustrates problems with the complexity of the software (Szczepański, 1987). The task was the following: write a parser of guidance commands for a combat navigation simulator in the Fortran. Each command could include several flight parameters and had to be parsed and executed in a single unit of simulation time. Back then, Fortran supported neither Object-Oriented Programming (OOP), nor sufficient abstraction layer to provide a generic solution for parsing such commands. The code was simple in principle, but the software fabric, which connected particular fields in command with flight parameters was so complex that it required more than ten pages of documentation and training on how to use it. Even after a few months after release, trivial errors such as wrong interpretation of a parameter were present in the code. In that case, complexity was enforced by the programming language limitations, but even nowadays, with all of the mechanisms such as polymorphism or template programming, such an inefficient and complex approach appears in the software from time to time.

(3) DRY

Rule No. 3: Do not apply the "copy and paste" technique during software development. It takes more time to correct it.

Another great practice is "do not repeat yourself." If there is a fragment of code which shares a similar structure or function, it is always a good idea not to copy it all over the software. Instead, such functions should be generalized or included in a common section and applied to a specific part of the software package.

Let us imagine two independent accelerometer-based subsystems. One of them is detecting acceleration for IMU/AHRS system, and the other one is a collision detector. If the accelerometers are of the same type, they might share configuration routines which will be then parametrized. You may need additional testing coverage for this procedure, but ultimately you do not have to write and test the same thing twice. In addition, you can reuse these routines in other projects or implementations. You may need only minor adjustments to make it work.

In the same way we can think about digital filtering or mathematics primitives such as matrix or quaternion multiplication. Such implementation may need a more extensive testing than straightforward implementation, but the time and robustness gained make it highly profitable in the long run.

It may positively impact not only the code size, but it also makes debugging and bug fixing easier. If one finds a bug in the common section, a single patch will fix it for every usage of that procedure in the software. This approach will also lead to more generic solutions, which will simplify the further development process.

From FuSa's perspective, such an approach has its drawbacks. Shared code must be compliant to the highest safety standard in which it is used. On the other hand, when you have e.g., a SIL-4 certified matrix multiplication library in use and something is not working, you will focus on the implementation details. At last, you will look for a bug in the library, as the SIL-4 is the highest level according to IEC 61508 standard.

(4) Scout rule

Rule No. 4: Do not keep unclear parts of the previously developed software. Clarify them when discovered and do not procrastinate this work.

One of the scout rules is to leave a camping place in a better condition than found at arrival. The same rule can be applied with common sense to software development. Here is an example. Every project has its coding style which has been specified by previous developers. Some of them are defined, such as K&R or GNU, but usually they are applied with slight modifications according to a developer's preferences. From a workflow perspective, not a particular style, but its consistency is a crucial factor. All style derogations force developers to make an additional effort to analyze and understand the code, which may significantly elongate even the simplest task.

In order to make a project pleasant to work on, it is always profitable to correct style flaws according to a commonly agreed standard. Small mistakes, such as typos, bad grammar in comments, or too long lines are also worth fixing at the moment when they are noticed. More significant errors usually have to be reported and dealt with accordingly. If something has got your attention, e.g. a lousy style habit, it is a good practice to ask co-developers about the reason why it has been applied. There is a big chance that they share the same opinion about it, and it has to be corrected in order to improve the code quality for further comfort of development.

(5) Modularity

Rule No. 5: Do not perform all functionalities in a one-stop-shop.

Divide an elephant into elementary pieces and make them work as one creature. Then you can easily change that creature's elements.

Flight control is usually a complicated and an entangled piece of software where applying the "divide and conquer" rule makes the development much more manageable. Keeping functionalities in a small (KISS) and functional oriented modules/layers gives more flexibility, makes the code easier to manage, solves repetitive naming issues, and helps to avoid unnecessary feedback.

Many early implementations of autopilot software suffered from a lack of separation between components, i.e. code responsible for communication was interleaved by processing, filtration, and compensation routines. It often caused difficulties with expanding functionality or finding a bug.

A much more efficient approach is to design modules and track interactions between them. In such structured software, the change in one part of the software should not affect the others. Modification of Hardware Abstraction Layer (HAL) will allow for e.g. transmission protocol change improved transfer efficiency by DMA or support of new autopilot hardware. By modifying raw data in the measurement routine, higher accuracy sensor support can be easily added. Significant changes, such as airframe type, may also require only a reassembly of the already present control modules and redesign of signal mixers. Such capabilities allowed e.g. Pixhawk or Ardupilot auto-piloting software to support many airframes and hardware vendors on a variety of platforms – from bare-metal microcontrollers (MCU) up to Linux-driven computers with features like cameras, Wi-Fi modules and similar.

One of the projects, which one of the authors participated in, was terminated due to a lack of sufficient API separation between the modules. The software was developed on Cortex-M3 MCU, without the Floating-Point Unit (FPU). On-board software used state-of-the-art solutions in terms of Digital Signal Processing (DSP) and navigation. Unfortunately, cost related to implementing floating-point operations became too high. It enforced the reduction of main loop frequency, which led to problems with stability margins.

Due to the complex nature of software, the author was

unable to port the whole software to MCU with FPU easily. A decision was made that only navigational and DSP parts will be moved to the Cortex-M4 processor on a dedicated extension board. One of the significant problems, which occurred afterwards was related to the program's structure. Separation of AHRS and navigation module required major changes in almost all the routines in Autopilot and Ground Station due to, among other things, complex calibration procedures.

(6) Defined APIs

Rule No. 6: Do not write an API without its precise specification before starting coding activity.

Good module separation cannot be achieved without a proper API specification. Multiple signals and data processing streams are crucial parts of the flight controller. If each component had a well-defined task, inputs, outputs, and functionality, this would significantly increase clarity of the system operation. Similar data streams can also be aggregated into a bus, which may be handled by e.g. a dedicated structure, enabling signal dependency tracking and simplifying a logging and telemetry transmission.

Ignoring such recommendations may lead to errors such as using raw instead of filtered data, creating unintentional loops in adaptation algorithms, or creating unnecessary cross-dependencies between software components, which may lead to problems with portability and multiplatform support.

(7) Recovery handling

Rule No. 7: Do not think that recovery of your software failure will never happen.

Plan recovery processes of the software failure most effectively and safely, even for the cases which cannot occur.

As Arianne 5 example has shown, in the control software all of the errors, even unexpected, should be handled safely. In FuSa terms, such behavior is called "failing gracefully" and is required or highly desired. The typical situation implies a fast recovery to the fully operational state. Usually, the "safe side" failure solutions are also acceptable. This goal may be achieved by exception handling, simplified backup algorithms, or data integrity checks. All unhandled erratic behaviors are potential points of failure, even if they are not supposed to occur in proper program execution.

Rule No. 8: Use safety features embedded in your platform — track status of execution.

Most modern environments have embedded safety mechanisms such as assertions, watchdogs. For some people, the time used for proper configuring them may seem like a waste of time. Do not be one of them! Let us imagine that you are developing e.g. a motor controller. In the safe implementation, you can monitor if control data is received in given intervals. What might happen if you do not have such a solution and communication cable will disconnect when the powerful engine is spinning e.g. a massive propeller at full throttle?

Such solutions are called using the Japanese term "poka-yoke", which means "error proofing." If you can improve safety using a build-in procedure, it is the most convenient way to do so. You meet them daily. Did you think of why in ATM you receive your credit card back before the money is given to you? There is one main reason. When you go to ATM, you are focused on the goal – getting the money. Forcing you to grab your card before getting money helps you not to forget a card.

In software development, it may have many flavors. You can assert that if the given data is valid, you can track the state of the data and order of procedure execution. It is also good to think about what purpose your implementation will be used for. You can then find the things that people forget or do wrong, and care about their safety even without their noticing it.

(8) Tests

Rule No 9: Do not think your software is ideal.

It needs extensive testing, starting from the simplest basic units of your code.

The industry standard of software development often embeds the Test-Driven Development (TDD) technique for tracking and avoidance of software bugs. The main reason for that is related to significantly lower costs in terms of time and effort to patch a bug at an earliest stage of its development. Finding a bug embedded into a code which has many software layers below and was written a long time ago is highly inefficient. It consumes much time to investigate the source of the error or may require contacting the original author.

A variety of tests preventing the situations mentioned above are defined. In order to verify correctness, acceptance criteria, and properties of a small piece of a coding unit may be applied to the performance tests. Integration and functional tests can be used to validate the cooperation of components in the final solution, which can then be verified against the requirements by the acceptance tests in a real environment using e.g. the Hardware-in-a-loop (HIL) technique.

In complex systems, changes embedded into one module may cause erratic behavior in another seemingly unconnected component. The leading cause of such errors is usually tough to track. In order to detect them early, periodical regression tests are desired.

Coverage and static analysis tests can also test the proper execution of the code. They can detect other types of hard to find bugs such as invalid conditions, variable overflow, or a dead piece of the code.

One of the benefits of extensive testing is enabling developers to apply one of the Extreme Programming rules: "Fail it until you make it." It is especially desired for complicated and hard to develop pieces of code. It can also assure a proper design of the recovery handling procedures.

(9) Four eyes principle

Rule No. 10: Do not think you are a perfect software developer.

Share your ideas and code with others. They can see what is unfeasible to be seen by you.

"Four eyes principle" is a crucial element of many modern, agile programming and management techniques. It implies that at least two people should make any significant decision. It helps to avoid errors caused by subjective bias. This useful rule can also be embedded in a software development process as a review requirement. A review process enables other developers from the team to share their thoughts and comments about the code changes before they are delivered into the mainline of software under development. If the team and the software architect accept the proposed changes, they are embedded into the common base for further development.

Such an approach not only improves error avoidance, but also allows development uniformity and consistent coding style across the entire project.

(10) Proper use of development tools

Rule No. 11: Do not mix the tested and reliable parts of software with the newly developed if the latter have not been appropriately checked.

Modern development support tools enable developers to ensure safe cooperation on their software. However, there are two significant aspects to be aware of, which cannot be ensured by even the most advanced solutions.

One of them is to keep experimental, untested code in separate branches as long as they are not ready to be embedded into the final solution. Hacks and temporal functional overwrites are often required during the addition of a new, sophisticated functional update. One of the most important things is not to forget about keeping them tagged and separate from the mainline of the development branch. Not only does it allow other developers to work on the functionalities separately, but it also ensures the safety of operation, e.g. during flight tests which use experimental software.

The other, no less important aspect is to track temporary workarounds and not yet implemented functionalities. Time pressure on software developers is usually significant. It often leads to forgetting about things which need to be done. Putting a "to do" comment in code or an issue in the task tracker may prevent situations when the solution only seems to be ready for delivery while being left unfinished.

(11) Take your time

Rule No. 12: Do not be in a hurry when writing software.

You need to prepare healthy slow food which supports the vehicle.

Getting things right is far more important than doing them fast, especially in the development of software for flying objects. A danger of causing personnel injuries or crashing UAV on a car or a building is always a worse option than facing consequences of delivery after the deadline.

4. SUMMARY

The proposed solutions and methods are just the tip of an iceberg in terms of FuSa and safe software development for UVs. Many experienced developers may find the presented guidance being only truisms, but neither of them, we hope, will disagree with the importance of the presented aspects nor will deny the existence of failures caused by ignoring the above mentioned rules. As long as failures caused by improper development techniques happen, there is a need for a public debate on the robustness of unmanned vehicles software. The number of crashed prototypes should not be the measurement of vehicle motion control advancement. Recent improvements in a code quality of the popular open-source flight controllers dedicated to UAVs, such as ArduPilot (ArduPilot Code repository, 2017), LibrePilot (LibrePilot Code repository, 2017) or Pixhawk (Pixhawk Code repository, 2017), have shown that the need begins to be noticed. All UV software programmers share the same goal: to make software as failure resistant as possible. It is the required step for breakthroughs in aviation of the future, such as autonomous flights over urban areas in the so-called U-space. Software already started to follow this path, but there is still much work to do. The same problems are connected with other types of UVs. They are not so broadly and loudly discussed as they are mostly being developed for professional applications, often military, or for use in unpopulated areas. In such cases, the development cost factors prevail the safety reasons.

REFERENCES

Ardupilot Code repository. Available at: https://github.com/ArduPilot/ardupilot, accessed on: 29 April 2019.

IEC 61508, 2010. Functional safety of electrical/electronic/programmable electronic safety-related systems. International Electrotechnical Commission (IEC), Sydney, Australia.

International Standard ISO/IEC 14882. Programming Languages – C++, ISO/IEC.

Leveson, N.G. & Turner, C.S., 1993. An investigation of the Therac-25 accidents. Computer, 26(7), pp.18–41. Available at: http://dx.doi.org/10.1109/mc.1993.274940.

LibrePilot Code repository. Available at: https://github.com/librepilot/LibrePilot, accessed on: 29 April 2019.

Lions, J.L., 1996. Ariane 5 – Flight 501 Failure, Report by the Inquiry Board. Available at: http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html, accessed on: 26 October 2017.

Mars Climate Orbiter Mission Homepage, 2000. Available at: https://mars.jpl.nasa.gov/msp98/orbiter/, accessed on: 29 April 2019.

Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A, RTCA DO-332, RTCA Inc.

Pieniążek J., 2014. Kształtowanie współpracy człowieka z lotniczymi systemami sterowania. Oficyna Wydawnicza Politechniki Rzeszowskiej, Rzeszów, Poland.

Pixhawk Code repository. Available at: https://github.com/PX4/Firmware, accessed on: 29 April 2019.

Sauser, B.J., Reilly, R.R. & Shenhar, A.J., 2009. Why projects fail? How contingency theory can provide new insights – A comparative analysis of NASA's Mars Climate Orbiter loss. International Journal of Project Management, 27(7), pp.665–679. Available at: http://dx.doi.org/10.1016/j.ijproman.2009.01.004.

Sommerville, I., 2015. Software Engineering, Addison-Wesley Publishing Company, USA.

Szczepański, C., 1987. Adaptation of application software for SL-106 simulator instructor stand. Report Optyka Ltd, Warsaw, pp. 43 (in Polish).