Fail-Safe Systems from a UAS Guidance Perspective

Christoph Torens and Florian-Michael Adolf
German Aerospace Center (DLR), Institute of Flight Systems, Department of Unmanned Aircraft, Braunschweig, Germany

1 INTRODUCTION

Unmanned aerial systems (UAS) are highly automated systems that will in the future populate segregated and also unsegregated airspace. The aerospace domain is safety critical and the utilization of such highly automated systems in safety-critical areas imposes high quality and safety standards onto these systems. As a result, manned aircraft are designed to be fail-safe to reduce risk of harm for the crew, passengers, or people on the ground. Of course, the same should apply for unmanned aircraft. The risk of harm for people should be minimized and a UAS should be designed so that if it fails, it fails safely. Conceptually, heterogeneous aircraft systems, for example, from the size of an insect up to the size of Global Hawk, all fall into this category. This work discusses the question of how unmanned aircraft can be made fail-safe when software complexity is growing and the human pilot is removed from the onboard system. For unmanned aircraft, there are several challenges to create a fail-safe system.

Questions are as follows: What does fail-safe mean in context of an unmanned system? How safe is safe enough for an unmanned aircraft? How can software-intense systems be made fail-safe?

The DLR is researching these questions with its ARTIS platform (Autonomous Research Testbed for Intelligent Systems), shown in Figure 1. Recent activities focus on verification and certification aspects, especially for the software modules that implement high-level behavior, for example flying in unknown terrain, and thus replaces the onboard human pilot (Torens and Adolf, 2015).

The remainder of this chapter is structured as follows: Section 2 describes the ARTIS research platform as background. In Section 3, related work is presented on fail-safe systems, especially software systems. The basics of fail-safe systems are presented in Section 4, discussing special features of UAS and software intense systems. Section 5 discusses the importance of comprehensive and correct requirements specification for the design and operation of fail-safe systems and shows how formal requirements were elicited for ARTIS. Next, Section 6 shows how software design influences system safety and proposes using formal methods for design validation. Verification and validation procedures are presented in Section 7. Section 8 discusses the important topic of runtime monitoring for unmanned aircraft and the influence of runtime monitoring for fail-safe systems. Finally, Section 9 concludes this chapter.

2 ARTIS SYSTEM DESCRIPTION

ARTIS represents a family of unmanned rotorcraft testbeds developed by the German Aerospace Center (DLR) for unmanned aircraft research purposes. Since 2006, ARTIS
has been equipped with a Mission Planning and Execution (MiPlEx) software framework that comprises real-time mission plan execution, 3D world modeling, and algorithms for combinatorial motion planning and task scheduling. The framework is based on a decoupled approach for path planning, trajectory generation, trajectory following, and inner loop flight control (Lorenz and Adolf, 2011). The rotorcraft’s guidance algorithm has been evaluated in flight tests with respect to closed-loop motion planning in obstacle-rich environments. (Adolf and Thielecke, 2007) describe a control architecture behind the guidance layer. It achieves hybrid control by combining the main ideas from a behavior-based paradigm (Brooks, 1990; Flanagan et al., 1995) and a three-tier architecture (Bonasso et al., 1996). The behavior-based paradigm reduces system modeling complexity for composite maneuvers (e.g., land/takeoff) as a behavior module that interfaces with the flight controller (Figure 2). The three-tier architecture has the advantage of different abstraction layers that can be interfaced directly such that each layer represents a level of system autonomy.

UAS operators have an intrinsic need for reduced mission planning complexity when specifying collision-free paths and mission tasks (e.g., search covering a given area). With the onboard control system, it is possible to design and execute mission plans. This planning system automates the translation of user-specified sets of waypoints into a sequence of parameterized behavior commands. The path planner is able to find collision-free paths in an obstacle-constrained three-dimensional space. The task planner determines a near-optimal order for a given set of tasks. Moreover, a task planner can solve specialized problems, for example specifying the actual waypoints for an object search pattern within an area of interest.

As a result, the mission planner must build three-dimensional, safe, and unobstructed paths quickly while optimizing task assignments and task orderings for multiple UAS, as shown in Figure 3. In the remaining work, this system will be used to demonstrate fail-safe systems development. The novel aspect of unmanned aircraft as an application is the missing onboard pilot, an aspect the public is most concerned about. Increased effort in verification and validation to achieve fail-safe guarantees are therefore required. We concentrate this chapter on this high-level software, the guidance component.

3 RELATED WORK

Even though safety requires a holistic approach, introducing software into safety-critical systems requires building safe software. A comprehensive analysis researching safety of software was done by Leveson and Harvey (1983). It defines software safety and describes a methodology to analyze software safety by using software fault tree analysis, an analogous technique to hardware fault tree analysis, with the goal of showing that the logic contained in the software will not produce safety failures. However, Leveson and Harvey (1983) state that safety analysis techniques must be combined with runtime-safety techniques, to catch errors or environmental conditions that are not recognized or anticipated during a priori safety analysis.
Figure 2. Mapping mission shown in context with the control architecture: High-level behaviors use task-specific planners (deliberate layer), behaviors are compiled into plans (sequencing layer), and movement primitives (reactive layer) interface with the flight controller. (Copyright © 2015 by DLR Institute of Flight Systems.)

Figure 3. Automated mission with nonlinear flight segments, a search area task, and a horizontal inspection circle. (Copyright © 2015 by DLR Institute of Flight Systems.)

Furthermore, Leveson (1986) specifically discusses the problems of introducing complex software into safety-critical systems. It is argued that many safety techniques developed for electromechanical systems do not apply when software is introduced. One reason is that these techniques often address random failures that are not a part of software. Traditional testing approaches cannot cope with this complexity, since testing cannot cover the complete state space. Furthermore, testing can only find errors according to the given specification. But wrong, misinterpreted, or inconsistent specifications can still lead to failures.

Leveson finally gives a general view about computer systems and approaches for achieving safety (Leveson, 1995). This book discusses risks, hazard analysis, safety require-
ments, and elements of a safe software design as well as verification aspects and also human factors. From a design point of view, there are two approaches to creating fail-safe systems: applying standards, codes of practice, and checklists into development and verification processes that reflect expert knowledge and lessons learned from previous accidents. The second is to use hazard analysis during design. For achieving the latter, there are four types of design techniques to achieve a fail-safe system: hazard elimination, hazard reduction, hazard control, and damage minimization. This technique should be used in the listed precedence order for a specific hazard.

Another overview of dependable computing systems is given by Geffroy and Motet (2002). A fail-safe system is described as a system that may fail, but when it does, either the failure consequences must not be “dangerous” or the probability of a “dangerous” failure must be smaller than a given acceptance level. As such, it is important for the design of a fail-safe system to define failure condition categories and set corresponding acceptable risks. Techniques to create fail-safe systems are based on intrinsic safety and safety designed by use of structural redundancy. Intrinsic safety utilizes a subset of technological development solutions that are known to be safe.

The work from Jhumka et al. (Jhumka et al., 2004; Jhumka and Sun, 2005) gives the theoretical background on generating fail-safe systems with a given safety specification. Any program specification can be considered formally as the intersection of a so-called liveness specification and a safety specification. They show that it is possible to generate a fail-safe program from a fault-intolerant program by using formal detectors for the safety specification. The generated, so-called, perfect multiliterior program, in the absence of faults, equivalent to the fault-intolerant program, but detectors remove transitions from the program’s state space that are inconsistent with the given safety specification. An overview of state-of-the-art and future research directions for safety-critical software engineering is given by Lutz (2000). Identified future research topics include integration of informal and formal methods, safe reuse of software, verification and runtime monitoring, education, and collaboration with related fields. Similarly, Knight (2002) discussed that development of safety-critical systems requires significant effort as well as advances in the areas of requirements specification, architecture, verification, and process management.

In summary, creating fail-safe systems requires knowledge from different areas, for example requirements specification, architecture, and verification. These are also the areas that are specifically discussed in this chapter (Sections 5, 6 and 7, respectively). Software failures are systemic; as such verification of software is crucial. However, software testing alone cannot cope with verification of complex software systems, since testing cannot cover the complete state space and not all environmental conditions can be anticipated. Therefore, the application of formal methods is discussed as a promising approach throughout the literature, especially for checking consistency with high-level requirements, as discussed in Section 5. Finally, runtime monitoring (Section 8) can be used to catch failures that arise despite careful software testing.

4 FAIL-SAFE SYSTEMS

Informally, a safety-critical system should not endanger people and thus should be fail-safe. Before discussing further details of fail-safe systems, it is necessary to provide a proper definition of this term.

A failure is an event that is inconsistent with the system specification.

A safety-critical failure is an event that is inconsistent with the system safety specification. A safety-critical failure can lead to a hazard and a potentially catastrophic event.

A fail-safe system is a system that is designed so that the occurrence of a safety-critical failure is mitigated in a safe way. Remaining system failures are either non-safety critical or the hazard probability is below an acceptable risk level.

It is worth mentioning that the definition of fail-safe systems does not claim that such systems do not fail. Instead, a system is expected to fail, but any system failure must occur in a safe way. Furthermore, fail-soft or fault-tolerant systems, which in case of a fault will still be able to perform most tasks, must not be mistaken for fail-safe systems. Fault-tolerant systems try to minimize all failures and not only safety-related failures, which also considers usability. A system can in fact be fail-safe but not fault-tolerant.

Fail-safe systems have a long and successful history. The automotive, railway, and aerospace domains are classical examples of mass-produced fail-safe systems. Due to its safety-critical nature, commercial aircraft have also been designed as fail-safe systems. The aerospace domain uses regulations and standards, for example DO-178C (Radio Technical Commission for Aeronautics (RTCA), 2011), ARP4754a (The Engineering Society for Advancing Mobility in Land, Sea, Air, and Space, 2010), ARP4761 (The Engineering Society for Advancing Mobility in Land, Sea, Air, and Space, 1996), and accreditation by federal authorities to ensure these principles. An aircraft is designed to be redundant; for instance, commercial aircraft have multiple engines. Furthermore, multiple controls and hydraulics
how fail-safe design techniques can be used as mitigation for hazards to ensure a target level of safety.

4.1 Fail-Safe UAS

A UAS is an aircraft with spatially distributed guidance by a pilot who acts on the physical world remotely. This requires onboard guidance to be fail-safe. We will specifically look at mission management and will highlight points that differentiate manned from unmanned aircraft. From a UAS perspective, the design of a fail-safe system is a challenge for several reasons:

1. The meaning of fail-safe for unmanned aircraft is not evident, since there is no pilot or passengers on board.
2. Strongly heterogeneous types and classes of unmanned aircraft exist.
3. There is no onboard pilot to complete specific tasks.
4. There is no onboard pilot to supervise specific tasks.
5. Software-intensive high-level functions are hard to verify.

We learned from Leveson (1995) and Geffroy and Motet (2002) that when speaking about fail-safe it is necessary to define an acceptable risk level for specific hazard categories. For a manned aircraft, for example CS25, this is strictly defined, and since people are onboard, a catastrophic failure has to be extremely improbable, which relates to a failure rate of $10^{-9}$. Some have proposed use of the same acceptable risk level for unmanned aircraft, since it might operate in the same airspace as manned aircraft. But, on the other hand, for an unmanned aircraft the risk of fatal casualties is lower. Only a midair collision with a manned aircraft or a crash in which two people are on the ground would result in fatalities.

Furthermore, consider the very different types of UAS. There exist efforts to create and use unmanned aircraft literally from the size of an insect, weighing only a few grams, up to the size of very large aircraft like Global Hawk, weighing thousands of kilograms. Between these extremes of unmanned aircraft, there is an almost continuous variation of unmanned aircraft sizes and also several different types of configurations. It would be very strict to assume the same risk categories for the multitude of unmanned aircraft, or to demand the same amount of rigor for a fail-safe system across all UAS. As a result, the acceptable risk should be assessed differently from that of manned aircraft. However, although efforts to create regulation for unmanned aircraft are increasing (European Aviation Safety Agency, 2015a; European Aviation Safety Agency, 2015b), there is no established definition for these acceptable risks yet.
UAS clearly differ from manned aircraft with respect to the pilot’s distance to the aircraft, his situational awareness, and his ability to bring resilience to unforeseen events into the guidance and control loop. The onboard pilot is replaced by a command and control data link that sends specified data from and to a pilot sitting at a ground control station. Because of these reduced capabilities, software has to take over functions that the pilot would usually perform onboard, resulting in a fully software system. For the sake of brevity, we will call this software component the guidance component of the unmanned aircraft. Writing and verification of software in safety-critical areas is already a problem for today’s aircraft. Additionally, it is possible to build unmanned aircraft that are a lot smaller, that is, no pilot can occupy the vehicle. But small systems, dependent on the actual size of the system, can be very limited in their capability to carry computing and energy resources. As a result, it is especially challenging to add redundancy to such systems. Another factor is that, due to the distance to the aircraft, the pilot has reduced situational awareness. Therefore, the pilot may notice problematic behavior of the aircraft only after it is too late to recover.

4.2 Fail-Safe Software

Hardware can be designed to be intrinsically fail-safe, for example trains use braking systems where pressure is constantly applied to open the brake and loss of pressure results in automatic activation of the brake. Therefore, a full stop, the safe state, for a train can always be ensured. In software (Leveson, 1986), different techniques are required and a safe state often cannot be entered from within the software. For example, software that uses dynamic memory allocation can behave unpredictably in case of error. Section 3 stated that software errors are strictly systematic. Special care is to be taken during software development and verification to reduce the probability that software errors remain in the final product. It is necessary to eliminate as many errors as possible, because any hazard that results from a software failure in fact was not detected during this activity. Therefore, the standard for developing software in the aerospace domain (Radio Technical Commission for Aeronautics (RTCA), 2011) is based on strict conformity to such development and verification processes and objectives. But on the other hand, software does not wear out, so no errors will be introduced during the system lifetime.

Furthermore, in contrast to hardware, it makes no sense to rely on a software only system. As long as the error is not fixed in the software design, the same erroneous software will reside in that system. The same problem arises, when trying to use redundancy to increase safety. To actually achieve redundancy, the same software cannot be used twice, instead software diversity must be achieved. The software needs to be completely designed, implemented, and verified a second or even a third time in independent development branches, essentially doubling and tripling the costs. Furthermore, each of these implementations is dependent on the quality of software verification activities to ensure that no errors exist. However, there are several approaches for using redundancy of independently developed software components. To realize fail-safe guarantees, an independent subsystem must be able to detect a failure in an active system. In principle, there either has to be some vote of $n$ out of $m$ systems, which is very costly and not trivial, or some kind of checks have to be performed on the output of the system via a runtime monitor. It is therefore possible to deactivate the failed system and switch to an independent system that re-establishes a safe system behavior, possibly with reduced capabilities. A more detailed depiction of software safety tactics is presented in Wu and Kelly (2004).

The remainder of this chapter shows the best practice approach used for ARTIS to achieve high software quality. Our approach uses three layers of safety: checking requirements and design using formal methods, a multitude of verification techniques with different properties for the software implementation, and runtime monitoring to find errors at runtime and enable fallback strategies. As mentioned, software standards like DO-178C focus on development quality and conformity to development processes. Although this is an important aspect of creating fail-safe systems, this is not discussed further in this work. For details on conformance of ARTIS development with DO-178C, refer to Torens and Adolf (2013).

5 FAIL-SAFE UAS REQUIREMENTS

The basis of fail-safe system design is a set of functional, nonfunctional, and safety requirements. These requirements are also used for verification and validation and runtime monitoring system design. Whereas the safety requirements are specific to fail-safe design, formally modeled safety properties could specify the behavior to ensure consistency and formally guarantee the intended behavior is possible, for example can always reach a safe state. Model checking is a technique to ensure the consistency and safety properties of such specifications.
Table 1. Requirements in a tabular semiformal format.

<table>
<thead>
<tr>
<th>ID</th>
<th>When</th>
<th>System</th>
<th>Obligation</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>When the safety pilot intervenes</td>
<td>the mission-manager</td>
<td>must</td>
<td>go into pause state immediately</td>
</tr>
<tr>
<td>6.1</td>
<td>When the UAS is landing and gets a stop command and the UAS is already on ground</td>
<td>the mission-manager</td>
<td>must</td>
<td>go to status on ground immediately</td>
</tr>
<tr>
<td>6.2</td>
<td>When the UAS is landing and gets a stop command and the height is above the safe minimum height</td>
<td>the mission-manager</td>
<td>must</td>
<td>go into StandbyAir mode immediately</td>
</tr>
<tr>
<td>6.3</td>
<td>When the UAS is landing and gets a stop command and the height is below the safe minimum height</td>
<td>the mission-manager</td>
<td>must</td>
<td>cancel landing immediately and rise to safe minimal height</td>
</tr>
</tbody>
</table>

Source: Copyright © 2015 by DLR Institute of Flight Systems

5.1 Requirements Formalization

A semiformal method was used to write down requirements for ARTIS. The tabular template was sufficiently easy to support directly writing down requirements, while it was formal enough to facilitate the use of keywords and a fixed structure. The template also was helpful in formulating the requirements and ensuring requirements were considered with all relevant aspects. The used template is shown in Table 1 and is an adaptation of a requirements template with conditions (Pohl, 2010). The disadvantages of using natural language for requirements can thus be minimized. Fields like "Activity" are used in this example mainly to fill natural language parts and the other fields, for example, "System" and "Obligation" are positioned to make the resulting row more readable.

After an initial learning phase, it was relatively easy to transform the tabular requirements to a formal specification in most cases. For the formal part, LTL (linear temporal logic) (Figure 4) and CTL (computation tree logic) (Figure 6) were used. It was easier to express conditions in LTL if possible. CTL seemed a bit harder to get used to. The used model checking tool NuSMV is able to handle both languages and the expressiveness of these languages is different, so both languages were used.

The examples show a clear correspondence of keywords on the left side of the property (left of the implication "→") and the "When" column of the tabular requirement specification, for example landing, stop, ground, MinHeight. The same observation can be made about correspondence of keywords on the right side of the property and the "Activity" column of the tabular requirement, for example, Standby-Ground, Slowdown, CancelStart. Furthermore, in most cases the requirement usually is globally true; therefore, the structure is $\text{LTLSPEC} \ \text{G}(\text{lefthand} \rightarrow \text{righthand})$. Not all, but a big part of the requirements could be handled...
8 Vehicle Design

Figure 6. Some liveness properties in LTL and CTL, that is, all states shall always be reachable and occur infinitely often in an endless run. (Copyright © 2015 by DLR Institute of Flight Systems.)

With this left hand, right hand scheme. The requirements shown in Table 1 are transformed with this process into a formal specification (see Figure 4). With the act of formalizing these requirements, the first objectives of DO-178C (Radio Technical Commission for Aeronautics (RTCA), 2011), respectively the formal supplement DO-333 (Radio Technical Commission for Aeronautics (RTCA), 2011), are already fulfilled, thus supporting safety (Torens and Adolf, 2015).

5.2 Safety, Liveness, and Fairness

In addition to translating the requirements into a formal model, some specific requirements are also to be modeled to ensure general model safety and liveness properties. In this context, a safety property is formally defined. A safety property essentially states that “something bad never happens” (Figure 5), whereas liveness properties state that “something good eventually happens” (Figure 6), more details for the definitions of these properties can be found in the respective literature (Clarke, Grumberg, and Peled, 2000; Baier and Katoen, 2008). These liveness properties are necessary to prove that the model is not trivial. For example, a UAS could never takeoff, then trivially all safety-critical requirements would automatically hold.

A fairness constraint further allows only those paths in the execution of the model that allow the fairness constraint to hold infinitely often. In this example the fairness properties ensure that in an infinite execution each command will be triggered (in a suitable state) infinitely often.

6 FAIL-SAFE SYSTEMS DESIGN

System architecture and design can affect the safety of a system in two independent ways. First, it can directly affect safety by facilitating or limiting safety as an integral part of the architecture. Furthermore, architecture and design have to be compliant to requirements, as required by software standards DO-178C (Radio Technical Commission for Aeronautics (RTCA), 2011). In this section we will discuss the former aspect briefly on the design considerations and revisit a similar approach in more detail later. The second aspect will be detailed by showing a prototype approach to verifying software design.

In Ref. Cherepinsky and Pinto (2012) a safety-critical architecture is shown, suitable for autonomous behavior of unmanned aircraft. High-level functionality executed with reduced criticality ensures support for sophisticated functionality and behavior. On the other hand, a second software module with higher integrity and priority can act as a backup with lower level, basic functionality to always ensure a safe state.
6.1 Design Verification

We utilize formal methods to check software design and facilitate model-based design approaches. For the creation of the formal model, at first a state diagram (Figure 7) was used to visualize thoughts and further clarify requirements as an intermediate step. Figure 7a shows the operational mode of the mission manager, including “safe” states standby_ground, standby_air, and a pause_state. The remaining states represent different kinds of movement, for example, landing. Figure 7b shows a separate task that can receive and validate missions transferred from the ground control station.

This model was then transformed into a formal NuSMV (Cimatti et al., 2002) specification. NuSMV uses a textual input language and allows definition of a behavioral model, that is, a finite state machine (FSM), and properties to be validated on the model in both CTL and LTL. A FSM can be specified via an assign declaration by giving an initial value init(var) and state transitions next(var), which are defined by boolean expressions. The different FSMs are then “executed” concurrently by NuSMV. A model-checking algorithm tries to find counterexamples for the properties that could occur during parallel execution of the FSMs. Eclipse was used as an editor, which featured syntax highlighting using an available NuSMV plug-in and a derivative plug-in distribution. Execution of model-checking could be done directly from within Eclipse using the external tools option.

The two FSMs of Figure 7 can be translated to NuSMV syntax straightforwardly. The transitions are guarded by a logical expression, for example checking actual state, and can then be executed by assigning the next value. To retain general comprehensibility, the produced expressions mimic a simple syntax: state + event = next-state. But in some cases, additional state variables further alter the behavior, as in the following example:

1 http://code.google.com/a/eclipselabs.org/p/nusmv-tools/
2 http://code.google.com/a/eclipselabs.org/p/nuseen/

Encyclopedia of Aerospace Engineering, Online © 2010 John Wiley & Sons, Ltd.
This article is © 2016 John Wiley & Sons, Ltd.
This article was published in the Encyclopedia of Aerospace Engineering in 2016 by John Wiley & Sons, Ltd.
DOI: 10.1002/9780470686652.eae1147
10 Vehicle Design

ManagerState=execute_mission & Command=mission_done & Position=air
:standby_air; --R3HL

This example shows that it is generally possible to translate a previously developed FSM into a formal model.

While building the model, the properties can be used to test it. Therefore, it is most important to have clearly specified requirements. Also, a complete traceability greatly supports a correct incremental build of the model.

6.2 Model-Checking Results

Conceptually, the model and its properties are developed independently. If the properties and the model are valid and the model also flawlessly represents the software, an error found by the model checker would always reveal a flaw in the software. In practice however, these parts usually debug each other mutually, as stated in the last section. The cause of an error may be one or a combination of a modeling error, a property formalization error, incorrect or missing environmental constraints, or errors in the requirements themselves. This is one of the key benefits of formal methods. It is therefore possible to find errors in the requirements while developing the formal model, which in turn can reduce development costs.

Temporal logic statements are evaluated as true for the specified model. A detailed discussion about conformance to DO-178C (Radio Technical Commission for Aeronautics (RTCA), 2011) and DO-333 (Radio Technical Commission for Aeronautics (RTCA), 2011) is discussed in (Torens and Adolf, 2015). However, it was invaluable to do the model checking to be able to validate the requirements themselves. A missing link still exists today between reality (code) and model checking. This gap is only filled by human expert review. Future work will enable assured conformance of the formal model and the source code or the object code.

7 FAIL-SAFE SYSTEMS VERIFICATION

Comprehensive system verification is a complex task; therefore, there are several layers of testing for the ARTIS system. Details for development and verification processes can be found in Ref. Torens and Adolf (2013). In this work, the MiPlEx testing efforts are discussed along with categories of testing. Here, static tests, dynamic tests, Software-in-the-Loop, and Hardware-in-the-Loop simulations, flight tests, and model-based tests will be elaborated as the different testing layers in an integrated test strategy. The presented verification approach utilizes these methodologies in a way that the tests start from inner, low-level aspects (source code level) and go to outer, high-level aspects of the system (integration and system level). Different test layers have different test characteristics and different costs. A good test design should combine these techniques to achieve a maximum coverage of all test characteristics. The key is to find errors at the earliest possible moment. This considers the fact that software errors tend to cause exponentially growing costs, the latter they are found during software development.

7.1 Test Methodologies

The first identified test dimension is the size of the specific system under test (SUT) that is used by the verification technique, for example what is being tested. Possible values range from single lines of code, software functions, a whole software module, interaction of different modules, and software systems up to the complete embedded system. Test effort (TeE) defines the costs of a specific test method. The test or scenario complexity (ScC) is low if a mathematical function is tested standalone, increases with combination and interaction of functions, and is highest if a complex scenario is tested or simulated. The coverage (Cov) assesses the theoretical state space that can be covered by the method or the code coverage if that is not applicable. The Feedback time (Fet) describes how much time it takes to get the result from the test back to the developer. A short feedback time is good, as it allows faster development cycles. Finally, the level of automation (Aut) describes if and to what degree a test can be automated. These test dimensions are evaluated for the ARTIS approach and finally visualized for comparison in Figure 8.

A system engineering process starts with acquiring requirements. If the models can be formalized to a substantial level, then formal approaches can be utilized to analyze that specification. Extensive sets of high-quality test cases can also be generated from a formal model.

There are several different techniques that can be identified by static analysis. For ARTIS, a large number of static assertions were added inside the implementation. This ensures a feedback to the programmer directly during compile time. In particular, BOOST (Boost Unit Test Suite, 2012) (C++ source libraries) concept checks are used, which define and test type requirements during compile time. Furthermore, static code checkers are used to assess the code quality. The tools CCCC (CCCC - C and C++ Code Counter, 2013),Cppcheck (Marjamäki, 2013), and CppLint (Google, 2013) are used for the ARTIS project. Also, Doxygen (van Heesch,
Fail-Safe Systems from a UAS Guidance Perspective

2013) warnings are used to ensure code quality by identifying missing comments.

Dynamic analysis refers to running the executable, or parts of it to perform tests. This means unit tests, but also coverage and memory leak tests. In particular, a lot of effort was put into creating an extensive set of unit tests. Units are being tested on various integration levels. As a last step, high-level functionality is tested, for example mission manager, mission planner, behavior sequences, the corresponding sequence controller, the flight mechanics, flight controller, and roadmap features.

Simulations can be categorized into Software-in-the-Loop (SIL) and Hardware-in-the-Loop (HIL) simulations. SIL simulations are used to test the software integration level, that is, the main parts of the software system as a whole, including the interfaces to other software and hardware components. For this purpose, a simplified software vehicle simulator has been written. The simulation can act in a variable simulation time, where calls are executed synchronously. This enables the MiPIEx framework to perform fast functional tests. Asynchronous execution of tests is possible to test real-time behavior. For the next test step, HIL tests are run on the actual flight hardware. This test setup additionally embeds the target platform, actuators, and sensor fusion into the tests. On this level, system integration is tested up to the actual hardware. The execution is done asynchronously to enable real-time behavior. HIL tests can be used to assess computing time and real-time performance. More details on the simulation framework, which is able to integrate different kinds of vehicles in various scenarios, can be found in Ref. Dauer and Lorenz (2013). Finally, flight tests in a controlled environment validate the system. The flight test represents a system test in the test taxonomy.

7.2 Test Dimensions

In this section we discuss test characteristics to provide a comparative view on the different testing methodologies that have different strengths and drawbacks. Static tests, unit tests, scenario tests, SIL/HIL, flight tests, and model-based tests and verification techniques are compared qualitatively by using an easy-to-use visual approach. Within a good test strategy, the different aspects of test methodologies should be considered. The testing methods are all integrated to identify strengths and weaknesses of the different approaches.

Figure 8 shows a visualization of the different ARTES test dimensions as a star diagram. The axes are size of the specific system under test (SUT), test effort (TeE), scenario complexity (ScC), state space coverage (Cov), feedback time (Fet), and the level of automation (Aut). The different test methodologies are for Figure 8a unit tests, SIL/HIL, and

---


This article is © 2016 John Wiley & Sons, Ltd.

This article was published in the *Encyclopedia of Aerospace Engineering* in 2016 by John Wiley & Sons, Ltd.

DOI: 10.1002/9780470686652.eae1147
flight tests. Figure 8b features static tests, model-based tests, and model-checking methods (MBT/MC) and again SIL for a better comparison of the dimension values. Note that a high test effort results in a low score value for test effort in the diagram. Low scores are in the inner region, high scores are in the outer region of the star diagram. This means, the larger the area covered by a test methodology, the better. Each testing methodology is given a unique color. The diagram has been split into two sections, to better visualize values. This diagram shows that testing greater artifacts for the SUT as well as increasing the scenario complexity is difficult and always increases test effort (and thus lowers the test effort score). Nonetheless, the SIL/HIL and flight tests cannot be replaced by other test methods, because they offer an increasing degree of fidelity (at increasing costs) that is not matched otherwise.

The HIL simulations are necessary test steps before going into flight test, but they require manual integration overhead. Notice that for ARTIS development, the SIL simulations are extremely useful, because they cover relatively large artifacts of the SUT at an excellent feedback time and can handle a wide variety of scenarios. Furthermore, these high-level tests are automated and the mentioned benchmarks can be utilized to assess the simulations, which gives invaluable feedback for development. The used static tests are extremely cheap, compared to other methodologies. Combined with the possible level of automation, static tests should be used, even if the achieved test complexity is low. Furthermore, studies show that the benefit of model-based techniques (Utting, 2005; Utting and Legeard, 2006) and model-checking (Clarke, Grumberg, and Peled, 2000) also comes from the necessity to create a formal model and thus find errors in specifications or requirements. Additionally, such techniques promise a systematic coverage of the state space and thus seem to be an interesting extension to the usual unit and scenario tests.

In the provided visualization, ranges for specific values cannot be visualized for different characteristics of a flight test. Also, the specific results of this analysis may differ from project to project. For example, unit tests can achieve a different degree of scenario complexity with different efforts. However, in comparison, the conclusion usually remains the same. System or flight level test, integration tests, and unit tests are all necessary test categories. The test dimensions show the different properties to understand the use of model-based approaches. In general, test automation has shown to be of utter importance in the ARTIS testing strategy. Automatic tests and tests with earlier feedback information are often utilized in addition or, whenever possible, as a substitute for manual tests. In the overall view, the additional effort is usually worth the earlier feedback and thus lowers effort for development. Furthermore, all the identified test dimensions are covered by the presented strategy. Still, several improvements are possible for the current test strategy.

8 FAIL-SAFE SYSTEMS RUNTIME MANAGEMENT

As mentioned earlier, some systems can be created that are intrinsically fail-safe. This means that a safe state will be entered as soon as a system failure has been detected. In the case of software, the system has to be monitored to check whether the system has failed or not. In earlier sections, we mentioned the importance of runtime monitoring for fail-safe systems. However, the monitoring approach is a work in progress for ARTIS, although some concepts have been shown (Dauer, Goormann, and Torens, 2014).

Generally, if the monitor assesses a system fault, then a mitigation action must be triggered. The monitor could either initiate an action by itself, reset, and reactivate the faulty system or deactivate it and activate a backup system. The backup system could have reduced capabilities but would just be able to maintain or ensure a safe state. As a result, the monitoring of systems and subsystems is the key approach to design fail-safe software systems. A very important property of a monitoring system is that of independence from the system under observation, so no problem that addresses the system under observation may also obscure the monitoring of this system. Otherwise, the monitoring would not add structural redundancy, and thus not make the system safer. The work from Refs Schumann et al. (2013); Reinbacher, Rozier, and Schumann (2014) introduces three additional principles that should be achieved by a runtime monitor: Unobtrusiveness, which means no influence or alteration of the actual runtime behavior of the system, should be imposed by the monitor, so that the system does not need to be recertified with the addition or alteration of the monitoring, responsiveness, meaning the monitor must be able to continuously supervise the system and deviations must be noticed within a defined interval, and finally realizability, which ensures the possibility to use the monitoring as a kind of plug and play system that can be easily adapted, changed, and introduced into a new system.

One important benefit of runtime monitoring is that it is possible to capture and formalize observed properties into temporal logic. This gives engineers a powerful tool for verification and debugging. Furthermore, it is possible to make assumptions of system design explicit, therefore such invalid assumptions can be found easily. Not every invalid assumption will cause a failure, because it can be masked (dormant) until specific additional conditions are fulfilled. Without monitoring, such inconsistencies would not be noticed. Since these formal requirements can be synthesized...
Further, existing approaches to make software systems fail-safe have been addressed. Two approaches complement each other: safety accomplished at design time through safe software design and thorough verification activities on one hand and detection of failures at runtime to enable appropriate corresponding mitigation strategies on the other.

Software failures are systematic failures. Therefore, verification activities play a crucial role to make a system safe. The use of formal methods is important to check system requirements and design. But these techniques are not yet common, even for designing safety-critical systems. Even then, formal methods are only one aspect of a holistic verification strategy, that combines complementary techniques. However, while comprehensive verification should always be the final goal, not all software failures can be found by verification activities, and not all environmental constraints can be anticipated. Therefore, runtime monitoring is a key aspect to make software-intensive systems fail-safe.

The ARTIS UAS research platform enables the development and test of autonomous software functions. We have used this platform to demonstrate the described techniques within our best practice approach for designing fail-safe systems. Our approach uses three layers of safety: The first layer is checking requirements and the design using formal modeling of requirements and model-checking techniques. To eliminate software errors, a holistic testing and evaluation approach for our mission planning and execution (MiPIEx) framework was presented as the second layer. The different testing dimensions were described with their different properties, such as cost and test coverage. Furthermore, the way they interact and complement each other was analyzed by decoupling the different test dimensions and by comparing results of this classification. With this argumentation, there is high confidence for the MiPIEx software to be correct, because there are a large number of tests run on different layers before progressing to the final flight testing. For remaining errors and environmental conditions that could not be anticipated, a runtime monitoring approach is used as a third layer. The monitor supervises high-level behavior, safety, performance, and situational awareness. Monitoring adds structural redundancy to the system without actually having to implement the function twice, because backup functionality can have reduced capabilities. However, monitoring must not be seen as simplified implementation, instead it adds a formal perspective to the system that is focused on safety. Therefore, monitoring can be a relatively inexpensive way to achieve fail-safe system design with the added benefit of a formal tool for debugging. The benefit is even higher, when formal properties can be reused from earlier phases of checking requirements consistency. Future work will focus on refining the approach and implementation of our monitoring framework.

9 CONCLUSION AND OUTLOOK

This chapter discussed the various challenges in the domain of unmanned aircraft to design and create fail-safe systems. Because of unfinished regulations, the near endless multitude of aircraft sizes, and the missing pilot onboard, the risk to harm people must be assessed different for unmanned aircraft. To determine if a system is safe, acceptable risks must be established. Without the above data, reasoning about safety of unmanned aircraft remains problematic. This situation will persist until corresponding regulations are finished.

Safety properties can be inferred from the system safety assessment. Functional and nonfunctional properties can be reused from the requirements analysis phase. If the requirements have been formalized for consistency checks with the model-checking tool, as it was proposed in Section 5, the formalized requirements can be directly reused. This reduces errors from implementing complicated monitoring properties and ensures conformance to the initial requirements. Situational awareness can further be categorized into three levels. The first level is perception, a comparison of input data with threshold values. For the second level, input data are combined with additional context. Finally, in level 3 projection, the data are projected into the future and warnings about upcoming hazards can be issued. This can be seen as a countermeasure to spatial separation of the pilot from the unmanned aircraft.

As a result, the MiPIEx software component takes over active tasks that an onboard pilot should normally perform. To complement this, the runtime monitor takes over the supervisory tasks of the onboard pilot. As such, the runtime monitor not only supervises mere functionality but also acts as an intelligent component that ensures high-level decisions and actions do not cause a catastrophic situation and are consistent with known environmental conditions.

Fail-Safe Systems from a UAS Guidance Perspective
As a result, we presented a theoretically founded best practice approach for holistic fail-safe high-level software of a UAS.

REFERENCES


Fail-Safe Systems from a UAS Guidance Perspective


