TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Flugsystemdynamik

Safe and Robust Automation of Aircraft and System Operation

Christoph Krause M.Sc.

Vollständiger Abdruck der von der Fakultät für Luftfahrt, Raumfahrt und Geodäsie der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs

genehmigten Dissertation.

Vorsitzender:	Prof. Dr.-Ing. Manfred Hajek
Prüfer der Dissertation:	1. Prof. Dr.-Ing. Florian Holzapfel
	2. Prof. Dr.-Ing. Mirko Hornung

Die Dissertation wurde am 06.07.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Luftfahrt, Raumfahrt und Geodäsie am 15.12.2020 angenommen.

Abstract

This thesis proposes novel methods in the field of safe and robust automation of aircraft and system operation. It is split into multiple parts covering a generic design, implementation, and testing methodology, an automation for flight control systems of experimental aircraft, and a maneuver injection for flight tests.

The created methodology provides the development basis for the applications and allows for a deterministic design, concise implementation, and comprehensive testing of state machines within the development environment. It is based on a hierarchical decomposition strategy, which is used during the design phase of the applications to minimize complexity and to optimize testing. Modeling guidelines are developed to support the implementation of the software with minimized opacity and maximized maintainability. Furthermore, an incremental bottom-up application of formal methods is proposed, which ensures effective testing and guaranteed system properties.

Administering the flight control loops of experimental aircraft is a crucial part of the flight control computer. It is performed by the developed flight control automation and includes various operational modes, which are based on a strategy to enable the switching between all control authority levels of the flight control system. The operational management concept is targeted at unmanned aircraft with multiple users, but can be used for manned aircraft as well and increases the awareness of the operator or pilot concerning the current operational mode. Automatically executed contingency procedures are used to mitigate the adverse effects of mission changes and malfunctions. They lower the human workload and ensure a continuous automatic operation.

Flight tests can be automatically executed by the developed maneuver injection. This software, which is part of the flight control computer, is used to generate maneuvers that are used for testing of actuator tracking, determination of controller performance, or for identifying flight dynamic models. The generic design allows for a free parametrization of the maneuvers without re-implementation. The dynamic generation of maneuvers and the flexible choice of injection points on multiple levels of the flight control loops enable a generic implementation and allow for a safe execution. Advanced features like automatic trim point capture and verification are integrated to allow for effective flight test campaigns.

To prove the real-life applicability of the developed solutions, they are tested on aerial demonstration platforms. The general validity is increased even more by testing the contributions on multiple aircraft with very different specifications. These include an unmanned demonstrator, two experimental platforms that can be operated manned or unmanned, with one and four seats respectively, and a 19-seat utility aircraft.

Zusammenfassung

In dieser Arbeit werden neuartige Methoden im Bereich der sicheren und robusten Automatisierung beim Betrieb von Flugzeugen und deren Systemen vorgestellt. Sie ist in mehrere Teile gegliedert und behandelt eine generische Entwurfs-, Implementierungs- und Testmethodik, eine Automatisierung für das Flugsteuerungssystem von Experimentalflugzeugen und eine Manöver-Einspeisung für Flugtests.

Die Methodik bildet die Entwicklungsgrundlage für die Anwendungen und ermöglicht eine deterministische Gestaltung, präzise Implementierung und umfassende Tests von Zustandsautomaten innerhalb der Entwicklungsumgebung. Sie basiert auf einer hierarchischen Aufteilung, die während der Entwurfsphase verwendet wird, um die Komplexität zu senken und das Testen zu optimieren. Richtlinien für die Modellierung werden erstellt, um die Implementierung der Software mit maximaler Transparenz und Wartbarkeit zu ermöglichen. Darüber hinaus wird eine inkrementelle Anwendung Formaler Methoden vorgeschlagen, die effektive Tests gewährleistet und Systemeigenschaften garantiert.

Die Administration von Flugregelschleifen in Versuchsflugzeugen ist ein entscheidender Teil im Flugsteuerungsrechner. Dieser wird von der entwickelten Flugsteuerungsautomatik übernommen und umfasst verschiedene Betriebsarten, die auf einer Strategie beruhen, auf alle Kontrollebenen des Flugsteuerungsrechners zugreifen zu können. Das Betriebsführungskonzept ist auf unbemannte Flugzeuge mit mehreren Nutzern ausgelegt, kann aber auch für bemannte Flugzeuge eingesetzt werden und erhöht das Bewusstsein des Betreibers oder der Piloten über den aktuell Betriebsmodus. Automatisch ausführbare Notfall-Verfahren werden eingesetzt, um die negativen Auswirkungen von Missionsänderungen und Fehlfunktionen zu mildern. Diese senken die menschliche Arbeitsbelastung und stellen einen kontinuierlichen und automatischen Betrieb sicher.

Flugtests können durch die entwickelte Manöver-Einspeisung automatisch ausgeführt werden. Diese Software, welche Teil des Flugsteuerungsrechners ist, generiert Manöver, um das Folgeverhalten der Stellantriebe zu testen, die Leistung der Algorithmen zu ermitteln oder um flugdynamische Modelle zu identifizieren. Das generische Design ermöglicht eine freie Parametrisierung der Manöver ohne Neuimplementierung. Die dynamische Erzeugung von Manövern und die flexible Wahl der Einspeise-Punkte auf mehreren Ebenen des Flugreglers ermöglicht eine generische Implementierung und eine sichere Ausführung. Erweiterte Funktionen wie die automatische Erreichung und Verifizierung des getrimmten Flugzustandes wurden integriert, um effektive Flugtestkampagnen zu ermöglichen.

Um die Anwendbarkeit der entwickelten Lösungen in der Praxis zu beweisen, werden diese auf fliegenden Demonstrationsplattformen getestet. Die Allgemeingültigkeit wird durch den Test auf mehreren Flugzeugen mit sehr unterschiedlichen Spezifikationen noch weiter erhöht. Diese umfassen einen unbemannten Demonstrator, zwei Experimentalplattformen, die bemannt oder unbemannt betrieben werden können und einen bzw. vier Sitze haben, und ein 19-sitziges Mehrzweckflugzeug.

Danksagung / Acknowledgment

Der Inhalt dieser Arbeit entstand größtenteils während meiner Zeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Flugsystemdynamik der Technischen Universität München, im Zeitraum von November 2013 bis Dezember 2019.

Daher geht mein Dank zuerst an *Prof. Florian Holzapfel*, für die Möglichkeit diese Arbeit auf Basis meiner Tätigkeit am Lehrstuhl anfertigen zu können. Zusätzlich möchte ich mich für die Unterstützung und fachliche Betreuung, die vielen Chance bei der Mitwirkung an neuartigen Projekten und die Möglichkeit für die vielfältige fachliche Weiterbildung auf internationalen Konferenzen bedanken.

Erkenntlich zeigen möchte ich mich bei *Prof. Mirko Hornung*, für die Erstellung des Zweitgutachtens und bei *Prof. Manfred Hajek*, für die Führung des Prüfungsvorsitzes.

Monica Kleinoth-Gross möchte ich meinen Dank für die freundliche und zuvorkommende Unterstützung bei allen Verwaltungsangelegenheiten aussprechen.

Übergreifend möchte ich mich bei allen Kollegen des Lehrstuhls bedanken. Durch den Zusammenhalt und eure Hilfsbereitschaft hat die Zeit bei mir einen bleibenden Eindruck hinterlassen. Besonderer Dank gilt dabei den Teams von *SAGITTA*, *ELIAS*, der *DA 42* und der *Do 228*. Durch die gute Zusammenarbeit haben wir in den Projekten viel erreicht, worauf wir noch lange stolz sein können. Außerdem geht mein ganz besonderer Dank an die Bewohner des besten Büros am Lehrstuhl, *MW3615*. Durch eure effiziente Arbeitsweise habt ihr zu einer professionellen Atmosphäre beigetragen, die ich nie vergessen werde.

Im speziellen möchte ich mich bei *Tuğba Akman*, *Christopher Blum*, *Tim Fricke*, *Christoph Göttlicher*, *Markus Hochstrasser*, *Andreas Kleser*, *Martin Kügler*, *David Seiferth*, *Philip Spiegel* und *Chong Wang* bedanken. Die fachlichen aber auch besonders die nicht-fachlichen Diskussionen mit flüssiger Motivation, die Pflege der Auslandsbeziehungen und weitere Aktivitäten werden mir für immer in guter Erinnerung bleiben.

Thanks to *Mrs. Teri Skipper Carter* and *Mrs. Cindy Dix Dennis* for proofreading this thesis and providing valuable comments as native speakers. Additionally, I would like to thank all members of my American family, who have had a great impact on my life and personal development since 2003.

Abschließend möchte ich mich ganz herzlich bei meiner Familie und im Besonderen bei meiner Mutter *Margrit*, meinem Vater *Konrad* und meinem Bruder *Nikolas* bedanken. Ohne eure Unterstützung, Motivation und Hilfe hätte ich meinen Lebensweg nicht gehen können und diese Arbeit wäre nie entstanden.

As a side note, I would like to thank all contributors of LaTeX, TeXstudio, MiKTeX, and Inkscape for making it possible to not use Microsoft Word.

Contents

List of Figures

List of Tables

List of Code Listings

Acronyms

AC-IJ	Actuator - Injection Switch
AC-IM	Actuator - Injection Module
AC-SW	Actuator - Switch
ACB	Actuator Clutch Box
ACE	Actuator Control Electronics
ADC	Air Data Computer
AF-IM	Auto Flight Control System - Injection Module
AF-OR	Auto Flight Control System - Override Switch
AFCS	Auto Flight Control System
AGL	Above Ground Level
AiL	Aircraft in the Loop
ARC	Air Risk Class
ATHR	Autothrottle
ATM	Automated Teller Machine
ATOL	Automatic Takeoff and Landing
ATP	Airline Transport Pilot
BDD	Binary Decision Diagram
BDP	Boolean Decision Procedure
BDS	BeiDou Navigation Satellite System
BLOS	Beyond Line of Sight
C2	Command and Control
CMS	Control and Monitoring System
ConOps	Concept of Operations
CPS	Cyber-Physical Systems
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSAS	Control and Stability Augmentation System
DCU	Data Concentrator Unit
DLR	German Aerospace Center
EASA	European Union Aviation Safety Agency
EC	European Commission

ECU	Engine Control Unit
EMC	Electromagnetic Clutch
EP	External Pilot
EPDL	External Pilot Data Link
EU	European Union
FBW	Fly-by-Wire
FC	Friction Clutch
FCC	Flight Control Computer
FCCA	Flight Control Clutch Automation
FCDL	Flight Control Data Link
FCS	Flight Control System
FCSA	Flight Control System Automation
FDL	Flight Data Link
FDM	Flight Dynamic Model
FMS	Flight Management System
FO	Flight Operator
FSD	Institute of Flight System Dynamics
FTMI	Flight Test Maneuver Injection
GCE	Gear Control Electronics
GCS	Ground Control Station
GLONASS	GLObal NAvigation Satellite System
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
GRC	Ground Risk Class
HiL	Hardware in the Loop
IAS	Indicated Air Speed
IL	Inner Loop
IL-IJ	Inner Loop - Injection Switch
IL-IM	Inner Loop - Injection Module
IL-OR	Inner Loop - Override Switch
IL-SW	Inner Loop - Switch
ILS	Instrument Landing System
INS	Inertial Navigation System
IOC	Input/Output Controller
IPM	Input Processing and Monitoring
IRS	Inertial Reference System
JARUS	Joint Authorities for Rulemaking on Unmanned Systems
LA	Loiter Automation
LASALT	Laser Altimeter
MAAB	MathWorks Automotive Advisory Board

MAG	Magnetometer
MC/DC	Modified Condition / Decision Coverage
MCP	Mode Control Panel
MDL	Mission Data Link
MFD	Multi-Function Display
MiL	Model in the Loop
MO	Mission Operator
MTOM	Maximum Take-Off Mass
MTSA	Methodology for System Automation
NAA	National Aviation Authority
NAV	Navigation System
OA	Operation Authorization
OP	Output Processing
OPV	Optionally-Piloted Vehicle
PGA	Programmable Logic Array
PIT	Periodic Interval Timer
RADALT	Radar Altimeter
SA	System Automation
SAE	Society of Automotive Engineers
SAIL	Specific Assurance and Integrity Level
SDL	Shared Data Link
SDV	Simulink Design Verifier
SiL	Software in the Loop
SlCI	Simulink Code Inspector
SORA	Specific Operation Risk Assessment
SP	Safety Pilot
SRB	Safety Relay Box
TA-SW	Trajectory Control / Auto Flight Control System - Switch
TC	Trajectory Control
TC	Type Certificate
TDL	Termination Data Link
TDR	Thrust Director
TG	Trajectory Generation
TG-SW	Trajectory Generation - Switch
TO/GA	Take-Off/Go-Around
TUM	Technical University Munich
UAV	Unmanned Aerial Vehicle
VLOS	Visual Line of Sight
WoW	Weight on Wheel

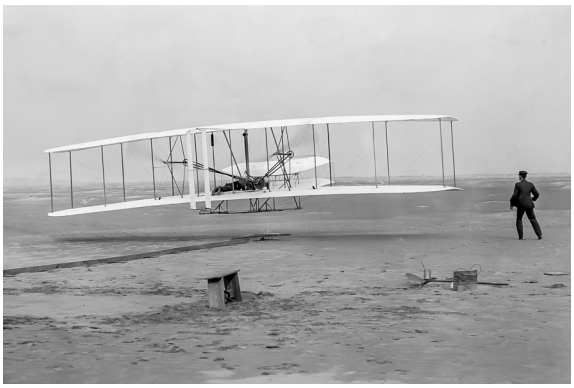
Chapter 1

Introduction

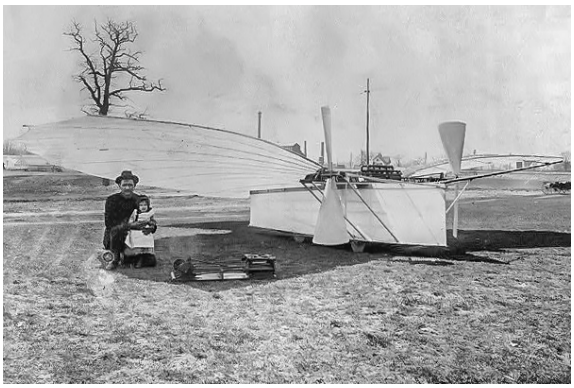
The history of manned aviation dates back to the very early 20th-century. To this day it remains controversial who performed the first sustained, controlled, powered heavier-than-air flight. The most widely recognized first flight was performed by *Orville Wright* on December 17, 1903, in North Carolina, with 37 meters [Smi2003]. Later that day his brother, *Wilbur Wright*, covered a distance of 260 meters [Wri1903b]. According to other sources, the first flight was actually performed, more than two years earlier, on August 14, 1901, by *Gustave Whitehead* in Connecticut [Jac2013]. His second flight on that date was allegedly over more than 2000 meters, far longer than the flights of the Wright Brothers.

Subfigure 1.1(a) shows a restored photograph of *Orville Wright's* famous first flight in Kitty Hawk, NC on December 17, 1903. A restored photograph of *Gustave Whitehead* and his No. 22 aircraft is shown in Subfigure 1.1(b).

Whether the first sustained, controlled, powered heavier-than-air flight was actually performed by Whitehead or Orville, mankind's dream of flying became true in the 20th-century. Only a few years later, aircraft were capable of flying many hours but still required continuous attention by the pilot. A system was needed to reduce the pilot's workload to ensure lower fatigue and safer flights for longer periods.



(a) Orville Wright [Wri1903a]



(b) Gustave Whitehead [Ive2015]

Figure 1.1: *Pictures of Wright and Whitehead*

(a) Sperry's Autopilot [GS2004] (b) Airbus A380 Cockpit [Mas2014]

Figure 1.2: *The First Autopilot and its Modern Descendants*

An automation of basic aircraft control function was developed by *Lawrence Sperry* in 1912, which became known as the first autopilot. It enabled the aircraft to fly straight and level with no interaction by the pilot [Amb2016]. A restored photograph of Sperry's Autopilot, equipped on the Curtiss biplane, is shown in Subfigure 1.2(a).

Modern aircraft have come a long way from those early days in aviation. One example, an Airbus A380 Cockpit, is shown in Subfigure 1.2(b). They require very little interaction and are capable to perform most maneuvers automatically. On the one hand, today's automation systems are increasingly used to enhance safety and economic performance [Alb1991]. But on the other hand, flying highly automated aircraft can lead to problems due to less experience with manual flight. The automation has made it more unlikely to encounter such problems, but at the same time made it more likely that the pilot will not be able to cope with them [Lit2019]. This has led to discussions about the advantages and disadvantages of higher-level automation in manned flight [Bai1983, Str2018].

In the field of Unmanned Aerial Vehicles (UAVs), manual flight is only possible in a limited range, due to data link latency, and would decrease the operational area. Additionally, operators need to focus on the mission's objective, not just the flying. Furthermore, without automation, failures like link loss would lead to the loss of the aircraft.

UAVs can be used in many different applications including precision agriculture, inspection of infrastructure, monitoring of wind energy, inspection of pipelines and powerlines, highway and traffic monitoring, monitoring of natural resources, environmental compliance, atmospheric research, media and entertainment, photos, filming, protection and research of wildlife, and disaster relief. [EAS2015b]

Performing such tasks only in manual flight would greatly limit the mission's effectiveness or even make it impossible in some cases. To exploit the advantages of UAVs, operators need to be able to focus on tasks, other than flying the aircraft. Therefore higher-level automation is indispensable for most UAVs. In the following, the motivation and background of this thesis are presented. Afterward, the state of the art, objectives, and contributions are shown before the remaining chapters are outlined.

1.1 Motivation

This section gives a short overview, of why UAVs have become more and more popular in recent years, what their advantages over manned aircraft are, what the current status of regulations is, and why a higher level of automation and system management is needed, compared to manned aircraft.

1.1.1 Popularity

In 1965 *Gordon Earle Moore* predicted that the component count in microelectronics will double about every two years, while the cost and size per component will be halved [Moo1965]. Since the invention of integrated circuits (the basis for modern microelectronics) in the late 1950s, this relationship proved to be correct and became known as *Moore's Law*. While it cannot continue forever and will probably stop working in the next few years [Cou2015], it shows how microelectronics have become smaller and lighter, while at the same time being more powerful and cheaper.

This progress is a key factor for the miniaturization and performance increase of digital flight control systems. The reduction of size and cost made it possible to use higher-level automation in smaller aircraft. Together with the demand for novel solutions, it has led to the rise of UAVs. Different configurations like fixed-wing, rotorcraft, multi-copter, or hybrid configurations like transition or tiltrotor aircraft have been used for various civil and military missions. Depending on the configuration and the mission, their size can vary between a few grams and several tons.

The development of the global UAV market from 2015 and the prediction to 2025 can be seen in Figure 1.3. During the last five years, the market has doubled and is predicted to continue growing even more in the future.

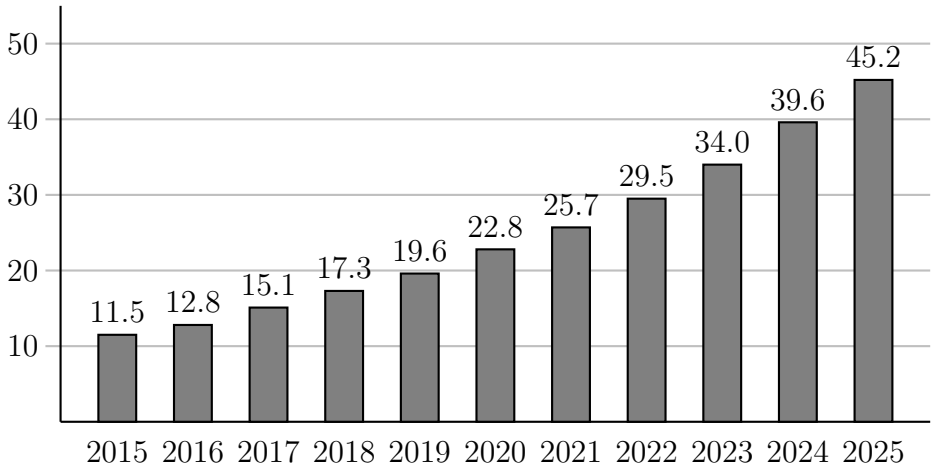


Figure 1.3: *Global UAV Market in billion USD (adapted from [Ame2017, Mar2019])*

1.1.2 Advantages

Whether or not UAVs have advantages or disadvantages over manned aircraft highly depends on the mission's objective, its duration, the environmental conditions, and other factors. The following list is summarized from [Aus2010, p. 5–7] and gives an overview of missions that can be more suitable for UAVs and reasons why to prefer an unmanned over a manned aircraft. The following proposed roles, where using a UAV is beneficial, are showing civilian examples. However, there are also corresponding military missions.

- Dull missions can be very tedious for the crew. Applications like extended surveillance can lead to decreased mission effectiveness due to loss of concentration. UAVs with modern ground stations can be more effective in such operations, with the additional benefit of possible crew changes for even longer missions.
- Hazardous missions like environmental monitoring for nuclear or chemical contamination as well as airborne crop-spraying with chemicals, expose the crew of manned aircraft to unnecessary risk. Using UAVs for such operations bypasses those problems with the advantage of easier detoxification of the aircraft afterward.
- Dangerous missions include powerline inspection or forest fire control, which also exposes the crew of manned aircraft to a high risk, which can be avoided when using an unmanned alternative.
- Covert missions, where the opposing side shall not be alerted, like following criminals, are very sensitive to low detectable signatures. Achieving this is easier with small-size UAVs, which produce less noise than their manned counterparts.
- Research projects with UAVs can be quicker, cheaper, and less hazardous than manned aircraft. Small-scale unmanned prototypes can be used for early testing of planned larger manned versions. In this stage modifications to the smaller version can be done more efficiently. Additionally, a novel configuration can be developed, which would not be able to contain a crew at all.
- Environmental reasons are related predominantly to civil missions. Due to its, most likely, smaller size and lower mass, a UAV needs less power which results in lower emissions and lower noise, compared to a manned aircraft performing the same task. This is especially beneficial for missions close to inhabited areas like powerline inspection or crop-spraying.
- Economic reasons, including acquisition and operating costs, will be important in almost any use case. Especially missions with smaller payloads will benefit greatly when using UAVs. As the payload gets heavier, for example in cargo flights, the percentage in the mass of the crew (including necessary onboard equipment) gets smaller and the benefits become less significant.

1.1.3 Regulations

In 2015 the European Union Aviation Safety Agency (EASA) acknowledged the diversity and innovative potential of unmanned aircraft and their associated industry. A regulatory framework was necessary to incorporate unmanned aircraft into the existing airspace. The proposed framework needed to be progressive- and operation-centric and follows a risk- and performance-based approach. [EAS2015a, p. 1]

This regulatory framework establishes three risk-based categories regardless of their maximum take-off masses (MTOMs), which are summarized in the following list. Ascending from low to high risk they are called: *open*, *specific*, and *certified category*. [EAS2015c]

- *open category* - low risk
  This category corresponds to "small" unmanned aircraft with an MTOM of $25kg$ and operation within Visual Line of Sight (VLOS) which poses a low risk to third parties on the ground or in the air. It relates most likely to hobbyists and will not require and authorization by a National Aviation Authority (NAA).

- *specific category* - medium risk
  Exceeding any of the limits of the previous category, requires specific mitigation of a higher risk, which is addressed in the *specific category*. This category allows operation out of VLOS and limited sharing of airspace if the risks are analyzed and mitigated through a Specific Operation Risk Assessment (SORA) and an Operation Authorization (OA) is issued by an NAA.

- *certified category* - high risk
  Certification will be required if the risk rises to a level similar to that of manned aviation like international cargo transport or transport of persons. In this category a Type Certificate (TC) for the UAV will be necessary, verifying that the design of the UAV is considered appropriate for different operations.

Due to the rapid development within the *open* and *specific categories*, the introduction of a framework for those was prioritized and resulted in a proposed amendment complemented by an impact assessment in 2017 [EAS2017a, EAS2017b]. In 2018 an opinion was published to mitigate the risk in the *open* and *specific categories* [EAS2018]. Based on this opinion, a proposal in 2019 was accepted by the EASA Committee and adopted by the European Commission (EC) [EAS2019]. Operations of unmanned aircraft within the *open* and *specific categories* shall be possible in 2020.

While the *open category* corresponds to hobbyists, where automatic operation is not required and the *certified category* raises requirements comparable to manned aviation, the novel *specific category* bridges the gap between the two. Operations beyond low risk are made possible without requiring certification and therefore opening a wide range of new possibilities. To exploit those benefits higher-level automation is necessary to extend the operational range and allowing the operator to focus on the mission's target.

1.1.4 Higher-Level Automation

For medium-sized UAVs, in the *specific category*, existing flight control systems, which have been developed for their larger counterparts, are not suitable. They use failure mitigation strategies that result in duplex or triplex system architectures to ensure continued operation or at least safe disconnection. This increases both size and weight of the equipment and makes the integration into smaller vehicles impossible in most cases. Additionally, the cost for those (certified) flight control systems makes them not feasible for smaller cost-effective platforms.

When operating a UAV, manual flight might not be desirable or even possible. On the other side full autonomous operation might also not be desirable, because it requires a high effort in the design and implementation of functions that will only be necessary for very specific situations. Table 1.1 shows different levels of automation for manned flight, as defined by *Charles Billings* [Bil1991, p. 27]. This table has been slightly modified from its original version, which is depicted in Appendix A as a comparison.

The table shows different management modes sorted by their level of automation or involvement by the pilot and the respective human and automation functions. The different levels range from *Direct Manual Control* with basically no automation and a very high level of human involvement all the way up to *Autonomous Operation* where the automation replaces the need for any human interaction.

Very low levels of automation like *Direct Manual Control* and *Assisted Manual Control* are only used in General Aviation nowadays. What is referred to as manual control in commercial aviation is described as *Shared Control* in Table 1.1. The next two levels of automation, *Management by Delegation* and *Management by Consent*, are the most commonly used modes of management in commercial aviation. Very high levels of automation like *Management by Exception* and *Autonomous Operation* are not used in today's commercial manned aviation.

To exploit the advantages of UAVs, as outlined at the beginning of this section, they are mostly used in *Management by Delegation* and *Management by Consent* modes. In some parts of missions also *Shared Control* or *Management by Exception* might be used. In those management modes, some functions that would normally be carried out by the flight crew need to be automated, to ensure safe and continued automation. Typical human tasks for manned flight that need to be automated for unmanned flight are emphasized in italics in Table 1.1.

They cannot be executed by the operator on the ground due to limited information, time constraints, or possible data link loss to the UAV. Additionally, monitoring functions process a lot of data, which might not be available in real-time to the ground crew. Therefore, time-critical decisions need to be made onboard because larger latencies make it impossible for the operator to react fast enough. Furthermore, decisions required as a consequence of communications failure, inherently cannot be handled by the ground crew and must be made automatically by the higher-level automation onboard the aircraft.

Table 1.1: *Levels of Automation (adapted from [Bil1991, p. 27])*

	Management Mode	**Automation Functions** / **Human Functions**	
Very Low ← Level of Automation → Very High	Autonomous Operation	Fully autonomous operation; Pilot not usually informed; System may or may not be capable of being disabled	**Very High ← Level of Involvement → Very Low**
		Pilot generally has no role in operation; Monitoring is limited to fault detection. Goals are self-defined	
	Management by Exception	Essentially autonomous operation; Automatic reconfiguration; System informs pilot and monitors responses	
		Pilot informed of system intent; *Must consent to critical decisions*; *May intervene by reverting to lower-level*	
	Management by Consent	Full automatic control of aircraft and flight; Intent, diagnostic and prompting functions provided	
		Must consent to state changes, checklist execution, anomaly resolution; Execution of critical actions	
	Management by Delegation	Autopilot and Autothrottle control of flight path; Automatic communications and nav following	
		Commands altitude, heading, speed, Manual or coupled navigation; *System operations and checklists*	
	Shared Control	Enhanced control and guidance, Smart advisory systems; Potential flight path and other predictor displays	
		In control through CWS or envelope-protected system; May utilize advisory systems; *System management*	
	Assisted Manual Control	Flight director, FMS, Nav modules; Data link with manual messages; Monitoring of flight path control and systems	
		Direct authority over all systems; FMS is available; Manual control, aided by F/D and enhanced navigation display	
	Direct Manual Control	Normal warnings and alerts; Communication with ATC; Routine ACARS communications performed automatically	
		Direct authority over all systems; Manual control with raw data; Unaided decision-making; Manual communication	

Additionally to the flight control algorithms, these novel monitoring, time-critical and control-level-change functions need to be integrated into smaller and less powerful hardware. Those functions need to be as safe as possible without relying on duplex or triplex system architectures. Furthermore, they also need to be robust with respect to deviations from the design model since modeling, system identification and flight testing need to be kept at a minimum to allow a cost-effective development.

Therefore, novel automation solutions need to be developed which can be used in cost-effective system architectures and are as safe and robust as possible.

1.2 Background

The objective of this thesis is to present the implementation and verification of automation functions. The developed methodology is applied to two functions, which are demonstrated on different platforms. Often used terms of the aerospace industry are described in the following subsection and the scope of the thesis is separated with respect to those. Additionally, the operational concept for demonstrating novel automation functions on experimental UAVs and Optionally-Piloted Vehicles (OPVs) is outlined. Furthermore, the development context and working environment, in which the methodology and automation functions presented in this thesis are developed, are introduced.

1.2.1 Aerospace Industry Terms

Within the aerospace industry, various terms with very distinct meanings are used. Therefore, the following section introduces the ones that are frequently used in this thesis or necessary for the overall understanding.

1.2.1.1 Manual Flight, Selected Flight, Managed Flight

In current state-of-the-art commercial airliners, three control levels are available. In *Manual Flight*, the pilot is controlling the aircraft using the control stick or yoke. However, in most cases, a computer is still supporting the pilot. When using *Selected Flight*, target values for the autopilot are commanded by the pilot through the Mode Control Panel (MCP). In *Managed Flight* commands for the autopilot are generated by the Flight Management System (FMS).

Others are working on bridging the gap between manual and automatic flight with an onboard artificial intelligence approach in cooperation with the human operator [KBS2011]. However, they are not designed for experimental aircraft and therefore not in the scope of this thesis.

1.2.1.2 GNSS, GPS, Galileo, GLONASS, BeiDou

Global Navigation Satellite System (GNSS) refers to a system that is able to determine the position of the aircraft. Global Positioning System (GPS) is the GNSS of the *United States of America*, which was originally known as NAVSTAR GPS. Galileo is the European GNSS, while GLObal NAvigation Satellite System (GLONASS) is operated by the *Union of Soviet Socialist Republics* and the BeiDou Navigation Satellite System (BDS) was developed by the *People's Republic of China*.

While the systems have their differences, there are all used for a similar reason. Since GPS has been operated for the longest and serves as a positing service for countless devices, it is used in this thesis representatively for all GNSS systems.

1.2.1.3 Aircraft Types

There are a variety of different self-powered aircraft types. The most commonly known is a fixed-wing aircraft, which, as the name implies, has a non-movable wing and one or more engines. Another widely used type is the rotorcraft, most commonly represented by the helicopter. In recent years multi-rotor aircraft also known as multi-copters have been designed in various forms and joined the rotorcraft family. Combining fixed-wing and rotorcraft has led to transition and tiltrotor aircraft, which can takeoff and land like a helicopter but can also fly efficiently for long times like a fixed-wing aircraft. The platforms used for demonstration in this thesis are all fixed-wing aircraft. However, the developed methodology and functions can be applied to other platforms as well.

1.2.1.4 Automatic vs. Autonomous

With the recent development in the automotive industry, the terms automatic and autonomous driving are used on many occasions. For driving automation, the Society of Automotive Engineers (SAE) has released a six-level standard for automation ranging from no automation (level 0) to full driving automation (level 5) [Soc2018]. Even though both autonomous and automatic are used in the aerospace industry as well, there is no such agreed-upon standard. Nevertheless, the seven Levels of Automation from *Charles Billings* can be used as a comparable counterpart.

While the terms autonomous and automatic are often used interchangeably, they are not. An autonomous aircraft is superior to an automatic one. The goals would be self-defined and it does not require any human input. To this date, there is no autonomous aircraft available. Therefore, this thesis is focusing on functions for automatic aircraft. Some even propose a more correct description of UAVs as uninhabited or remotely piloted aircraft [KH2005, p. 19].

1.2.1.5 Manned, Unmanned, Optionally-Piloted

Manned aircraft always have a crew onboard to control the aircraft directly or to set targets for the automatic flight control system and supervise it. In contrast, a UAV, only has a ground crew to set targets and in some cases supervise the automatic flight control system. If the aircraft is capable of both manned and unmanned flight, it's called OPV. Due to legal restrictions in the last years, it was nearly impossible to operate a medium- or large-size UAV in Germany, or Europe in general. Meeting required safety standards for unmanned flight would have resulted in increased cost and more engineering time, which was not possible in the projects. To bypass those problems, OPVs were used, which are capable of performing fully automatic unmanned as well as manual manned flight. However, in the scope of this thesis, OPVs are only used with an onboard safety pilot, who has the legal responsibility. Nevertheless, the functions developed in this thesis can be used for both, UAVs and OPVs.

1.2.2 Automation of UAVs and OPVs

The overall scope of this thesis is to present the automation of experimental aircraft based on the classical paradigm of *Supervisory Control*. In contrast to automatic control, which focuses on including all necessary capabilities for a specific task into the automation system, *Supervisory Control* includes the human operator at the highest level of abstraction. In this scenario, the system and human operator work together somewhere in the spectrum between manual and automatic control. This leads to a hierarchical relationship, in which tasks are separated between the human and the system in different degrees depending on the situation. The operator can focus on monitoring and mission objectives, while the automation system is controlling the aircraft. [She1992]

In the following, the operational concept of experimental UAVs or OPVs, as used in this thesis, is presented. Those platforms are used to demonstrate the real-life applicability of the methodology and automation functions presented in this thesis. The concept is depicted in Figure 1.4 and includes relevant participants and system boundaries between humans and the automation.

The operational concept can be divided into three segments, the *Ground Segment*, *Air Segment*, and *Space Segment*. The *Ground Segment*, which in some cases is also referred to as monitor- or supervisory-segment, contains the operator and/or user. The *Air Segment* includes the UAV or OPV and the *Space Segment* contains the GNSS satellites.

In the following, the three segments and the links between them are introduced in more detail. Depending on the demonstration platform, the operational concept can be slightly different, which is addressed in Chapter 2.

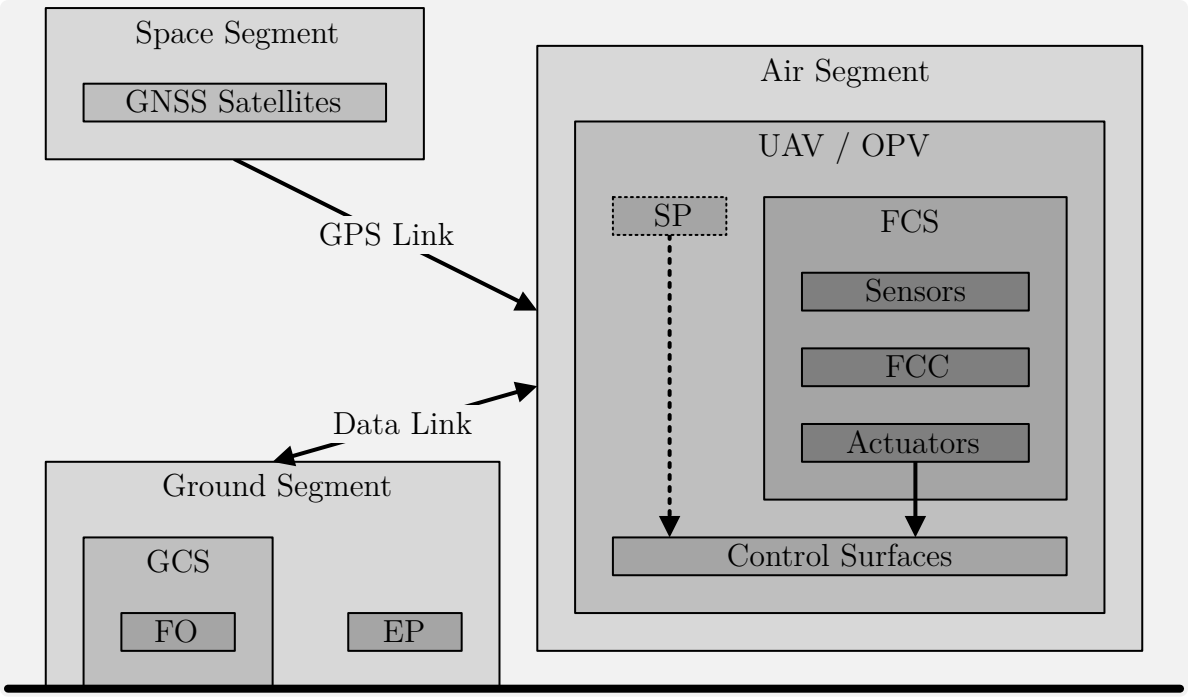


Figure 1.4: *Operational Concept*

1.2.2.1 Ground Segment

The operator, who is controlling the aircraft with higher-level functions is called Flight Operator (FO) and located within the Ground Control Station (GCS) in the *Ground Segment*. Possible commands that can be used are autopilot values (like altitude, heading, and speed) or a selection of different waypoint lists. Additionally, higher-level automation functions like autoland, return to base, or loiter can be activated. The FO is also responsible for the monitoring of the automation system on a mission level.

Depending on the demonstration platform one or more additional External Pilots (EPs) are located in the *Ground Segment*. In contrast to the FO, the EP can control the aircraft using low-level functions like attitude or rate commands. The primary reason for the EP is risk mitigation during the first testing phase of the experimental aircraft.

In a more conventional use case or mission, there is most likely a Mission Operator (MO), who is controlling the equipment for the specific mission. However, due to the close relationship of the FO to the aerial platform during experimental flights, this is not the case, since the flight tests are designed for the aircraft itself.

Due to the spatial separation, one or more data links are used for communication between the *Ground Segment* and *Air Segment*. In the case of the FO, it is a two-way link, which is used for transferring commands to the aircraft but also for feedback from the aircraft to the GCS. The data link for the EP is usually a low latency one-way data link used to control the aircraft within VLOS.

1.2.2.2 Air Segment

The UAV or OPV constitutes the *Air Segment*. In the case of the OPV demonstration platforms, a Safety Pilot (SP) is legally responsible for the aircraft and the pilot in command, while the Flight Control System (FCS) is automatically controlling the aircraft. An important feature of all OPVs is the ability to disable the FCS at any time, to return control to the SP. In the case of a UAV, there is obviously no pilot onboard the aircraft.

The FCS includes the sensors, the Flight Control Computer (FCC) as well as actuators to control the surfaces of the aircraft. Additionally, the OPVs are equipped with clutches between the actuators and components of the mechanical flight control system, to disconnect the automation in case of manual flight. The methodology and automation functions presented in this thesis are all part of the FCC.

1.2.2.3 Space Segment

The *Space Segment* contains only the GNSS satellites. An abstract representation of the processing of signals from multiple satellites by the GNSS receiver onboard the aircraft is shown as "GPS Link" in Figure 1.4. If the GNSS is not able to determine the position of the aircraft, this is referred to as GPS loss. In the thesis at hand, satellite-based communication is not considered and therefore omitted in Figure 1.4.

1.2.3 Development Context

The development context in which the methodology and automation functions presented in this thesis have been developed is depicted in Figure 1.5. For a better understanding, a short overview of this methodology and the developed functions are presented here.

To implement higher-level automation functions a methodology is necessary that makes the implementation of a multi-level finite state machine possible while retaining maintainability, reducing complexity, and permitting testing. One of the two main functionalities developed in this thesis, which uses this methodology, is a system automation, which is capable of administering a cascaded flight control loop with respect to commands from different operators or automatically due to sensor information and/or hardware integrity. Additionally, a maneuver injection module is implemented, which is capable of injecting predefined test sequences to various injection points within the cascaded control loop.

To demonstrate the real-life applicability of the developed methodology and automation functions they are tested on four demonstration platforms. Those are a $150kg$ UAV, two OPVs, and one manned aircraft. The OPVs consist of a four-seat twin-engine aircraft and a one-seat fully electric ultralight. The manned aircraft is a twin-turboprop aircraft with up to 19 seats. More detailed information about those platforms and their operational profiles, used for demonstration in this thesis, is available in Chapter 2.

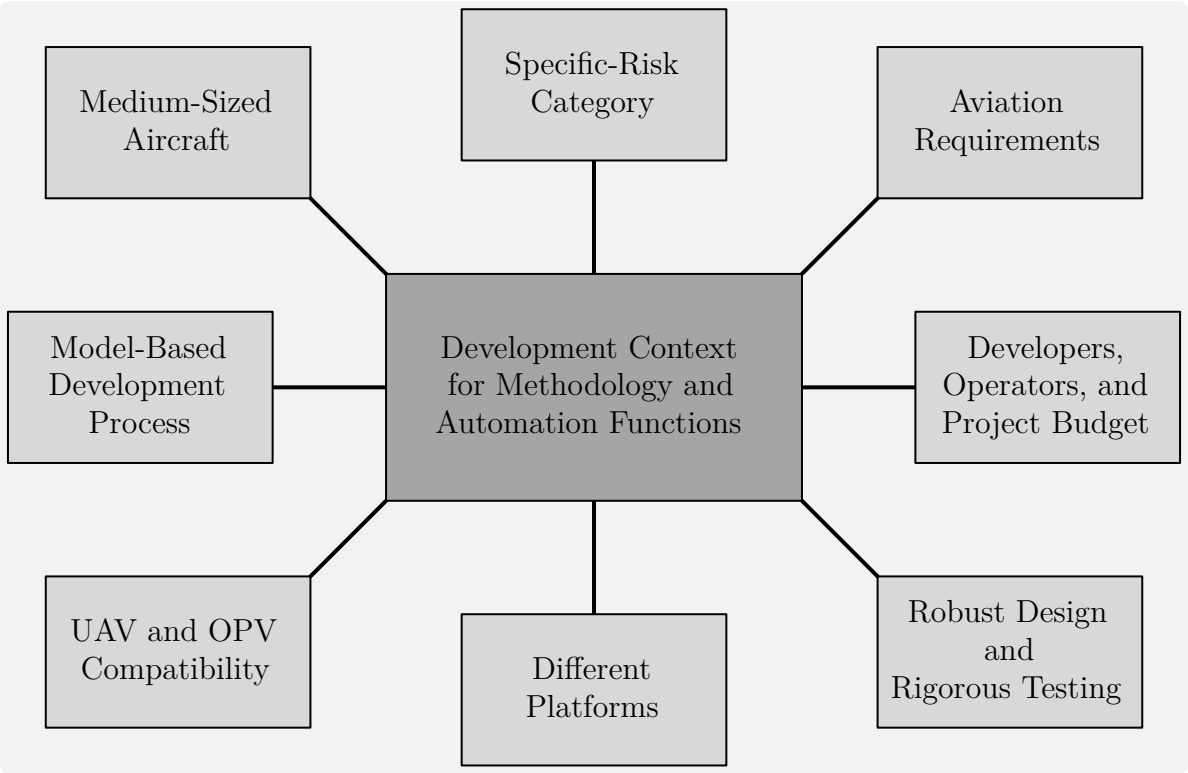


Figure 1.5: *Development Context*

1.2.3.1 Medium-Sized Aircraft

The target platforms for the developed software range from the smallest having an MTOM of $150kg$ up to the largest having almost $6000kg$. Even though they have quite a different mass and size, the architecture is very similar and they are, in this thesis, referred to as *Medium-Sized Aircraft*. With regard to hardware redundancy, they all have a simplex system architecture. Additionally, due to project cost constraints, size, and/or weight margins, they all have a limited sensor performance, quality, and expected integrity.

Despite the simplex architecture, the lower system performance, and the smaller integrity, safety is ensured through different measures. In case of a hardware failure in an OPV, the pilot can take over control in any situation. In the case of a UAV, the flight is taking place over a specific area and within closed airspace. Therefore, the loss of the aircraft can be accepted in case of a hardware failure.

Due to the diversity of the automation, cascaded control loops are necessary for the developed software modules. The automation needs to intervene on different levels, therefore the separation of control loops needs to be explicitly visible. The control loops of the cascaded controller include, among others, a waypoint generation and control module, an autopilot with autothrottle, and an inner loop for control and stabilization.

1.2.3.2 Specific-Risk Category

All platforms, used for demonstration in this thesis, would probably be assigned to the *Specific-Risk Category* representing a medium risk if they were used as UAVs within the European Union (EU) and without special permits for closed airspaces.

The Joint Authorities for Rulemaking on Unmanned Systems (JARUS) have defined guidelines on SORA, which support the application of risk assessment for authorization to operate a UAV within the *specific category*, which is introduced in Subsection 1.1.3 and further defined in [EAS2018]. The SORA process uses the Concept of Operations (ConOps) to derive a Ground Risk Class (GRC) and Air Risk Class (ARC). The GRC is based on the aircraft's dimension and kinetic energy, while the ARC is based on the used airspace during the mission. Those risk classes are then consolidated into a Specific Assurance and Integrity Level (SAIL), which drives the required activities for the authorized operation of a UAV. [JAR2019]

At the time of writing this thesis, formal requirements for the *specific category* are being developed. Therefore, this work is inherently designed to support the proving of deterministic properties. This includes the execution of test cases, Modified Condition / Decision Coverage (MC/DC) analyses, and formal verification of key functionalities. Demonstration flights, which prove the real-life applicability of the methodology and automation software developed in this thesis, were performed under "Permit to Fly" or within closed airspaces. Therefore, a certification with respect to any category or authorization has not been performed and is not part of this thesis.

1.2.3.3 Aviation Requirements

Additionally to the guidelines in Subsubsection 1.2.3.8, widely accepted *Aviation Requirements* are taken into account. Those requirements are integrated into both, the development of the methodology as well as into the implementation of the software functions. A key requirement of all safety-critical software with human interaction is a deterministic and unambiguous behavior. Additionally, its functionality must be in accordance with requirements, so that the described functionality is deterministically available. The automation must have guaranteed properties in line with the requirements, where the same input or series of inputs always leads to the same results. This enables the pilot or operator to be able to anticipate the reaction of the automatic system in a specific condition, which is the basis for a purposeful utilization of the system. Without this deterministic predictability, the pilot is not able to use the system in a meaningful way.

It also must be possible for the pilot or operator to intervene at any time. In the case of an OPV, this is not limited to changing commands but also includes taking over control and completely deactivating the system. When controlling a UAV, the deactivation of the system is obviously not desirable. Nevertheless, a smaller lever of intervention must also be possible. Furthermore, the system must be able to monitor itself and ensure a safe operation. It should even be capable of performing automatic contingency maneuvers, if pilot or ground crew reaction time is too long or not possible due to a connection failure.

1.2.3.4 Developers, Operators, and Project Budget

The development context and working environment with respect to the *Developers, Operators, and Project Budget* is significantly different than in classical aviation projects. Due to the scope of experimental UAVs and OPVs, the team is significantly smaller. Nevertheless, their tasks include the same areas of design, implementation, and testing.

The personnel actively involved in flight tests are usually not long-time trained pilots, like Airline Transport Pilots (ATPs). This includes the FO, EP, and flight test engineers. The only exception within the operating team of the aircraft is the SP, who is a specially trained test pilot.

The project budget, with respect to finances, is significantly smaller and the development time is much shorter. Due to the experimental nature of the project, their time frame is usually limited to a couple of years. Classical aviation projects on the other hand are designed for several decades.

Despite all those differences, the objective of the experimental UAV and OPV projects, referenced in this thesis, is an equivalent safety level. This means development, implementation, and testing methods need to be adapted to be suitable for this changed context. The model-based development process discussed in Subsubsection 1.2.3.8, helps to achieve those goals.

1.2.3.5 Robust Design and Rigorous Testing

The methodology also needs to allow for *Robust Design* and implementation with respect to complexity, brittleness, opacity, and literalism.

As software gets more and more versatile it also gets increasingly more complex. This can have negative effects on design, implementation, and testing as well as interacting with the software from a functional side like a pilot or operator. The software must be capable to perform the required task or mission, while at the same time keeping the complexity at an acceptable level.

Brittleness in software design refers to algorithms that appear to work reliable under normal circumstances but fail completely in unusual conditions. One goal of the developed methodology is therefore to avoid brittleness.

Another problem of complex software occurs when the monitoring pilot or operator does not understand why the algorithm is performing a certain reaction or isn't sure about the software's intention. This lack of transparency is also referred to as opacity. The implemented software should, therefore, follow easy-to-understand design decisions to enable the operator or pilot to easily develop an understanding of the behavior.

Robustness also requires avoiding literalism, which defines software that continues normal operation in unusual conditions. This should be avoided in this thesis by monitoring sensor integrity and performing automatic contingency procedures if necessary. This also allows for a reduced workload of the operator in challenging situations.

Rigorous Testing is necessary to ensure the completeness of testing, which in turn allows for a high confidence level of the correct functionality. This includes two aspects, which are, the software having the intended functionality while at the same time not having unintended functionality. The most basic requirement is, therefore, that the developed methodology must allow for this kind of testing.

The first step always includes a standalone software test to ensure key functionalities are working as designed. To get repeatable results it is necessary to generate test cases, which can be executed automatically. This includes using formal methods to find hidden design errors, analyze test case coverage, and prove certain requirements. Coverage analyses, for both statement coverage as well as MC/DC, are used to make sure the functionality is developed in a proper way.

Software testing also includes Model in the Loop (MiL) and Software in the Loop (SiL) simulations to check compatibility and functionality with other software modules and the Flight Dynamic Model (FDM). Testing the developed software on the real hardware is the next step. This is done in Hardware in the Loop (HiL) and Aircraft in the Loop (AiL) environments, where as much as possible of the real hardware is taken into account to verify the correct functionality on the same hardware as used for the flight test.

Ground Tests then prove the applicability of the software in a real-life environment with all other components, before, finally, Flight Tests are used to verify those results and to test the complete functionality of the software.

1.2.3.6 Different Platforms

The implemented functions must be applicable to *Different Platforms*. That means they need to be transferable and embedded into different software architectures, which requires a general interface, both on the input and output. Furthermore, the software needs to be parameterizable without changing it internally.

Additionally, the structure needs to have mode guards or be designed in a specific way that an unsafe mode for a specific aircraft cannot be activated under any circumstances. With those requirements, the software implemented in this thesis can be safely used on various platforms. The four platforms, the software is designed for and tested on, are described in Chapter 2.

1.2.3.7 UAV and OPV Compatibility

The designed software needs to have *UAV and OPV Compatibility*. Therefore the implemented modules need to account for different types of control inputs. This can be an onboard pilot who generates commands using a joystick or MCP. Since those inputs reflect two completely different levels of involvement, they need to be treated appropriately.

The pilot can also be represented by a human with a remote control or joystick on the ground. When the aircraft is in sight of the pilot, control modes can be similar to onboard ones. Due to higher latency when flying Beyond Line of Sight (BLOS) and/or with video feedback, associated control modes are most likely higher-level ones.

Another user can be the FO in the GCS. In this case, commands to the flight control system might include mode changes, adjusting command values for the autopilot, or changing target waypoints.

1.2.3.8 Model-Based Development Process

The methodology developed in this thesis is part of the *Model-Based Development Process* developed at the Institute of Flight System Dynamics (FSD). It is based on aerospace and automotive industry standards which helps to maintain high-quality software models and also includes formal verification [HHH2016].

Developing software modules as design models is part of this development environment. The developed methodology and implementation of software functionality described in this thesis are following guidelines, which are also part of the development process.

MATLAB, *Simulink*, and *Stateflow* are used for the design, implementation, and testing of the developed software [TM2020b]. The automation functions, described in this thesis, are largely based on decision logic and therefore implemented in *Stateflow*. Part of this thesis focuses on extending and enhancing those guidelines with respect to the used software for implementation and the developed module structure. The adhered to and developed guidelines used in this thesis are explained in more detail in Section 3.3.

1.3 State of the Art

While the functions presented in this thesis can all be categorized as automation functions for UAVs and OPVs, they are still quite different. Therefore, after giving a short introduction to the state of the art of aircraft automation functions in general, this section is split up with respect to the methodology and each of the two applications.

Even though the first automation functions were introduced shortly after the first manned flight at the beginning of the 20th-century and have since then increased dramatically, up to this day, there is no commercially available manned aircraft capable of performing higher-level automation functions like fully automatic flight from takeoff to landing. While those features are not so important for manned flight, because the pilots can perform those tasks, it is mandatory for unmanned aircraft to take advantage of their inherent benefits outlined in Section 1.1.

In recent years large military UAVs have been used for various missions. Necessary functions for those missions also include those, developed in this thesis. However, due to the military context, detailed information on the used methodology, design, implementation, and testing of functions is generally not available, but they would probably not be in the scope of this thesis anyway.

Medium and small-sized, with respect to size, weight, and budget, UAVs and OPVs gained attention in recent years, as technology has become more affordable and available. However, integrating higher-level automation functionality requires a modern system architecture and a significant amount of development effort. This is due to the complexity of the larger systems that cannot easily be transferred to a medium or small platform, because of the lack of space, weight reserves, and redundancy.

As a consequence, only a very limited number of test platforms, even fewer for manned aircraft, included the demonstration of those higher-level automation functions in real-life flight tests. Exceptional requirements like fully automatic maiden flight, have so far only been accomplished with large UAVs and their advanced flight control systems.

The goal of this thesis is to present the design and implementation of different higher-level automation functions, that utilize existing medium- and low-level control functions. To support those a methodology for finite state machines is developed, that is utilized by the two applications which are presented in this thesis. The system automation is able to administer a cascaded control loop with respect to commands from multiple users. Additionally, the automatic flight test maneuver module is designed for the accurate execution of numerous maneuvers that are fed into various injection points.

This section, about the state of the art, the next Section 1.4, containing the derived objectives, and Section 1.5, introducing the main contributions of this thesis are divided into three parts each. In this section, Subsection 1.3.1, Subsection 1.3.2, and Subsection 1.3.3 present the current state of the art with respect to the methodology and the two applications. In those sections and throughout this thesis, colored boxes with increasing color saturation are used to summarize the motivation, objectives, and contributions.

1.3.1 Methodology for System Automation

Automation of higher-level flight control system functions has been used in large military UAVs, which include the Northrop Grumman RQ-4 Global Hawk [Nor2001], the Boeing X-45 [Boe2002], the Dassault Aviation nEUROn [Das2003], and many more. However, they have MTOMs of several tons, estimated significantly higher budgets, and probably advanced and redundant system architectures. Therefore, they are not in the scope of this thesis and their automatic capabilities cannot be compared with those developed here.

Integrating that functionality into smaller UAVs or OPVs requires a significant amount of development effort and a modern system architecture. The developed methodology presented in this thesis supports the integration of complex higher-level automation functions, which are in general only available on those advanced large UAVs.

The implementation of multiple command modes on different authorization levels requires large state machines. Within the scope of this thesis, even more modes for experimental operation need to be addressed by the state machine, making it more complex than currently available solutions.

The decomposition of finite state machines to accomplish various objectives has been used for a long time. Especially in the early days of Programmable Logic Arrays (PGAs), it has been used to increase performance, due to the reduced path between inputs and outputs [ADN1992]. However, decomposition strategies within flight control software have not been published.

Formal methods for design, validation, and verification of higher-level automation for UAVs and OPVs are relatively new [LKN2011]. One example is the Formal Verification of the FCS 5000 [MAW+2005], where the FCS is modeled in *Simulink* and the formal verification is done using the New Symbolic Model Verifier. Application of formal techniques and model-based safety assessment in the field of state machine-based decision logic for higher-level flight control system automation using *Stateflow* has not been reported.

The motivation, resulting from the shortcomings in the current state of the art, with respect to the Methodology for System Automation, is summarized in the following list. Derived objectives are presented in Subsection 1.4.1.

Motivation - Methodology for System Automation

- A design and implementation approach for the development environment is necessary to reduce the complexity of higher-level automation
- Specific constraints must be met to suit the smaller development team, shorter project duration, future extension, and re-use
- Testing systematic based on formal methods is required to ensure safe and robust automation during experimental operation of UAVs and OPVs

1.3.2 Flight Control System Automation

Human-centered aviation automation has been considered in the classical automation of aircraft since the 1990s. Problems of accidents and incidents during that time were mostly attributed to the *Loss of State Awareness* associated with complexity, the coupling of functions, autonomy, and inadequate feedback. [Bil1996, p. 5ff]

Despite discovering those problems a long time ago, recent accidents and incidents still show similar problems caused by *Loss of State Awareness*, which is also referred to as *Mode Confusion*. When operating highly automated UAVs or OPVs, *Mode Confusion* is even more likely due to the physical separation of the operator and the aircraft.

As mentioned in the previous subsection, there are three main control levels of commercial airliners. However, their autopilot can only access modes on one control level and changes to another control level can only be made by the pilot. The only exception is the occurrence of sensor errors, which can lead to an automatic degradation to *Manual Flight*. This behavior is particularly critical because the automation is disabled in circumstances when it is needed the most by the pilots. This has also lead to several accidents and incidents in the past including Air France Flight 447 [Bur2012]. In contrast to commercial airlines the software of experimental aircraft, in the scope of this thesis, needs to be able to automatically switch between all available control levels and modes.

In the field of experimental UAVs and OPVs, manual and fully automatic flight needs to coexist and be available to the operator or pilot. Due to the novelty of this field, only a very limited number of publications related to system automation of experimental UAVs or OPVs are available. Within the FlySmart-FBW23 project, a Diamond DA42 is used to perform automatic takeoff and landing [PSJF2016, Dri2016, Rei2017]. A Stemme S15 is used in the LAPAZ project for automatic landing [DLR2013]. However, the automation of both systems is only designed for one user, has no contingency procedures and publications provide little information on software architecture, implementation, and testing.

The motivation, resulting from the shortcomings in the current state of the art, with respect to the Flight Control System Automation, is summarized in the following list. Derived objectives are presented in Subsection 1.4.2.

Motivation - Flight Control System Automation

- Higher-level flight control system automation is necessary to exploit the inherent benefits of UAVs and OPVs
- Currently available solutions are not suitable for experimental aircraft with multiple users
- No integrated contingency procedures are available in existing approaches, that support the operator in case of a failure

1.3.3 Flight Test Maneuver Injection

Flight tests have been used since the early beginning of aviation to prove the functionality and performance of aircraft and systems. They have also been used to build and verify FDMs, evaluate controller performance, and test actuator dynamics.

Normally, test pilots are used to manually conduct those tests on manned or optionally-piloted aircraft. However, for more complex maneuvers and if high precision is required, this is not possible. Additionally, the information gained from tests can efficiently be increased, when they can be automatically executed very precisely and repeated numerous times. Therefore, automatic flight test maneuver generation is highly favorable even for simple maneuvers and inevitable for more complex maneuvers.

Maneuver autopilots, that are capable of performing specific maneuvers have been used for development for a long time. Various publications are available, that discuss autopilots for flight tests of high-performance aircraft. Despite being not in the scope of this thesis, those maneuver autopilots have been designed for specific, mostly high angle of attack, maneuvers only and are not capable of injecting various types of signals to different parts of the control loop. [DJB1986, DHB+1988, MWD1989, HTA1991]

Others have focused on optimizing inputs to the control surfaces with respect to exciting the aircraft's response. Only multi-sine maneuvers have been used to control the surfaces to gain efficient information for aerodynamic analysis of model aircraft, and have not been used on manned aircraft. [Mor2012a]

In recent years similar approaches have been used for parameter identification of OPVs. Step- and ramp-maneuvers are used to control the surfaces and to perform aircraft identification maneuvers. However, various maneuvers, injections to other parts of the control loop, and demonstration of the software on manned aircraft are not included. [KHT2013]

The motivation, resulting from the shortcomings in the current state of the art, with respect to the Flight Test Maneuver Injection, is summarized in the following list. Derived objectives are presented in Subsection 1.4.3.

Motivation - Flight Test Maneuver Injection

- Automatic flight test generation is highly favorable for simple maneuvers and inevitable for more complex maneuvers
- Numerous highly customizable maneuvers and various injection points are not supported by current solutions
- Recent developments do not incorporate advanced functions for safe and efficient flight testing

1.4 Objectives

The objectives and challenges for the methodology and two applications are outlined in this section. They are derived from the current state of the art and resulting motivation, as presented in Section 1.3.

The overall objective for all parts of this thesis is the real-life applicability. The developed methodology is applied to the automation functions, which are used on real-life UAV and OPV demonstration platforms. In contrast to applications that are only tested in simulated environments, this poses a possible danger to humans. Therefore safe and robust design has the highest priority. Additionally, the developed applications must be usable on UAVs and OPVs without re-implementation.

The development environment at FSD is based on products from *MathWorks*, which are developing mathematical computing software [TM2020a]. *MATLAB* is a proprietary programming language and numerical computing environment, which serves as a basis for many toolboxes [TM2020c]. The toolbox used for flight controller design and implementation is *Simulink*, which is a block diagram environment for model-based design [TM2020d]. Additionally, *Stateflow*, a modeling environment for decision logic, is used to design and implement state machines representing operational modes of the aircraft [TM2020e]. *MATLAB*, *Simulink*, and *Stateflow* are used in combination throughout all parts of this thesis.

The methodology and applications developed in this thesis must be applicable within the general conditions at FSD and the development context. This includes effective solutions that can be managed by a small development team with a limited budget, efficient approaches that are compatible with the shorter project durations, and applications that can be operated not just by highly trained individuals.

The methodology is the conceptional part of this thesis. It is then used to design and implement the two applications. Their main features are summarized in the following list.

- The methodology for deterministic design and implementation of advanced finite state machines in *MATLAB*, *Simulink*, and *Stateflow* with reduced complexity and increased manageability
- The human-centered flight control system automation for multiple users, administering the cascaded control loop of experimental UAV and OPV featuring continued automatic operation in non-nominal circumstances
- The automatic flight test maneuver injection module for accurate execution of complex sequences supporting numerous maneuvers and various injections points with advanced features for efficient test campaigns

In this section, Subsection 1.4.1, Subsection 1.4.2, and Subsection 1.4.3 present the derived objectives of each part. The developed contributions are introduced in the next section, Section 1.5.

1.4.1 Methodology for System Automation

The objectives and challenges of the Methodology for System Automation are outlined in the following. They are derived from the current state of the art and resulting motivation, as presented in Subsection 1.3.1.

Higher-level flight control system automation is necessary to exploit the inherent benefits of UAVs and OPVs as explained in Section 1.1. The overall objective of the methodology of this thesis is to enable the implementation of complex higher-level automation state machines in *MATLAB*, *Simulink*, and *Stateflow*. This is necessary to design real-life applications within the development context described in Subsection 1.2.3.

Another important goal is the manageability of complex systems. This includes the design process, the implementation as well as the testing, and the operation of the software. The design and implementation are based on finite state machines using *Stateflow*. Deterministic behavior is very important in aviation and ensures a safe and predictable behavior of the automation.

Testing flight control software is very important. However, as described in Subsubsection 1.2.3.4, the smaller team, limited project budget, and reduced time are not able to support large test series like in commercial aviation. Nevertheless, an adequate safety level is required to ensure the robust functionality of the software during experimental flights. This must be supported by the methodology, by creating the basics for complete testability, even of complex state machines. This guarantees relevant system properties and requires less effort than classical test methods.

Furthermore, the methodology should promote expandability and transferability. The expandability of the automation by integrating new features needs to happen rather quickly due to the shorter project time as explained in the development context. Since the automation functions are used on multiple platforms, the methodology also needs to support the transfer to another system without re-implementation and re-testing.

The objectives of the Methodology for System Automation are summarized in the following list. The developed contributions, which are devised from them are introduced in Subsection 1.5.1.

Objectives - Methodology for System Automation

- Develop a methodology to implement advanced higher-level automation finite state machines in *MATLAB*, *Simulink*, and *Stateflow*
- Ensure the manageability of the system during design and implementation, focusing on future extension and usage on other platforms
- Implement a deterministic system that ensures a safe and predictable behavior as well as complete testability for robust and guaranteed properties

1.4.2 Flight Control System Automation

The objectives and challenges of the Flight Control System Automation are outlined in the following. They are derived from the current state of the art and resulting motivation, as presented in Subsection 1.3.2.

The overall objective of this software module is the automatic administration of control loops of experimental UAVs and OPVs. In the context of this thesis, the overall control of the aircraft is split up into multiple cascaded control loops. The administration module, presented in this thesis, needs to be able to access all of these control modules. The switching between control modules and modes needs to be transient free and the operational concept needs to be applicable to both UAV and OPV. This requires the representation of different use cases within the same system without re-implementation.

Mode confusion is still responsible for many incidents and accidents in recent years. To counteract the even higher probability of such problems in the context of highly automated experimental UAVs and OPVs, the developed software module shall be operator-centric. Because the operators controlling the aircraft in the context of this thesis are not highly trained, a predictable and easy-to-understand automation is even more important.

As stated in Subsection 1.3.2, current solutions only support one type of operator. In the context of experimental UAV and OPV operation, multiple users are involved in the execution of a mission (c.f. Subsection 1.2.2). Therefore, the administration module needs to provide interfaces for the different types of operators and users.

The execution of missions in the context of experimental aircraft and their limited system and sensor availability is more unpredictable than in the commercial environment. Most commercial systems revert to *Manual Flight* in case of malfunctions and even the current state of the art systems don't include contingency procedures. To increase the robustness of the automation in non-nominal operational circumstances and reduce the workload for the FO and EP, automatic procedures shall be implemented that support uninterrupted automatic control, even in presence of sensor or system failures.

The objectives of the Flight Control System Automation are summarized in the following list. The developed contributions are introduced in Subsection 1.5.2.

Objectives - Flight Control System Automation

- Implement a system automation to administer a cascaded flight control loop of experimental UAVs and OPVs
- Provide a operator-centered automation with interfaces for multiple users, like operators and pilots
- Develop procedures for robust automation, even in non-nominal operational circumstances, targeting operator workload reduction

1.4.3 Flight Test Maneuver Injection

The objectives and challenges of the Flight Test Maneuver Injection are outlined in the following. They are derived from the current state of the art and resulting motivation, as presented in Subsection 1.3.3.

The main objective of the flight test automation, described in this thesis, is the automatic and accurate execution of complex-signal flight test maneuvers. This is especially useful for maneuvers that cannot be executed by a test pilot with the required precision and/or repeatability. With multiple automatic replications, the validity of the results can be proven. At the same time, the workload of the pilot is reduced, who can then focus on monitoring for improved safety.

Most current solutions don't offer multiple maneuvers at all and the ones that do, only have a very limited number of maneuver types. Therefore, the module, presented in this thesis, shall support the integration of various maneuvers, while keeping complexity to a minimum. This is especially important when implementation is split up between multiple developers and is used on different platforms.

Another drawback of available solutions is that the injection is limited to the control surfaces only. The flight test injection is part of the FCC and its cascaded control loop and will be used for identifying aircraft and actuator dynamics as well as analyzing controller and aircraft performance. To unlock this full potential, injection points are not only needed on the surface level but also on higher controller levels.

Due to the context in which those functions are developed, more specifically the Model-Based Development Process (Subsubsection 1.2.3.8) and the Developers, Operators, and Project Budget (Subsubsection 1.2.3.4), efficient and accurate flight testing is especially important. To support fast and precise flight testing advanced features are necessary to improve the overall effectiveness of the project.

The objectives of the Flight Test Maneuver Injection are summarized in the following list. The developed contributions are introduced in Subsection 1.5.2.

Objectives - Flight Test Maneuver Injection

- Provide an automatic and accurate execution of complex-signal flight test maneuvers and the integration within the FCC
- Support the integration of numerous maneuvers and various injection points within the cascaded control loop
- Develop advanced features for safe and efficient flight testing, like individual trim points for each maneuver

1.5 Contributions

This section introduces the contributions beyond the current state of the art, which have been devised to meet the objectives presented in Section 1.4. They are divided into the generic methodology and the two applications. Those two functions are designed and implemented before being tested on multiple demonstration platforms to prove their real-life applicability. The interaction of the contributions is depicted in Figure 1.6.

The developed contributions are introduced in the following subsections. This includes the individual contributions in each field and a summary of their novel improvements beyond the state of the art. A short overview is presented beforehand. All contributions are verified by testing their functionality in various real-life flight tests on multiple platforms.

The *Design, Implementation, and Testing Methodology for System Automation - using State Machines in Stateflow (MTSA)* is a generic methodology for higher-level system automation that includes design, integration, implementation, testing, and verification. It enables deterministic, predictable, and robust automation of safety-critical software with guaranteed properties.

The *Operator-Centric Multi-User Flight Control System Automation for experimental UAVs and OPVs with Contingency Procedures (FCSA)* administers a cascaded control loop consisting of various modules and enables multiple users to switch between control modes. It also features contingency procedures to reduce the workload of the operator and to perform automatic maneuvers in case of a malfunction.

The *Multi-Maneuver Multi-Control-Level Flight Test Maneuver Injection with automatic Trim Point Capture and Verification (FTMI)* is capable of generating various types of test maneuvers and injecting them into different control levels of the cascaded control loop. Additionally, it includes individual trim point verification and even automatic trim point capture enabling effective and reliable flight testing.

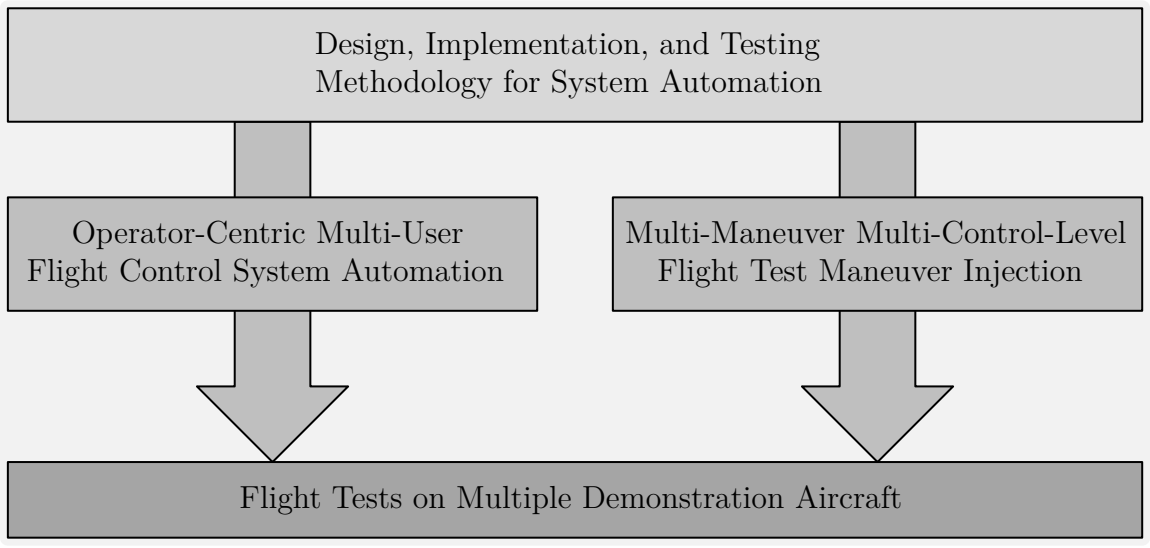


Figure 1.6: *Contributions*

1.5.1 Methodology for System Automation

The MTSA is devised from the objectives presented in Subsection 1.4.1. The developed contributions, beyond the current state of the art, are introduced in the following and explained in more detail in Chapter 3.

1.5.1.1 Hierarchical decomposition design strategy, minimizing complexity and optimizing testability

A novel hierarchical decomposition strategy is developed and presented in this thesis, which minimizes the complexity of higher-level automation state machines and optimizes testability for deterministic behavior. The strategy is based on breaking down the overall functionality into modular units. This enables separate development, implementation, and testing. Each unit consists of a state machine and an input- as well as output-conditioning. A few of those are grouped into one implementation level which is administered by another unit on the next higher level. This leads to a tree structure with various modes on different levels, which is implemented in a higher-level automation state machine with minimum complexity utilizing *MATLAB*, *Simulink*, as well as *Stateflow*.

1.5.1.2 Modeling guidelines for implementation, minimizing opacity and maximizing software maintainability

The new modeling guidelines for the implementation of automation that are presented in this thesis are targeted at minimizing opacity and maximizing software maintainability. The application of those guidelines creates an explicit visibility of action alternatives within the development environment. Additionally, they minimize the local variable scope of state machines aiding an easy understanding of individual state machines and their interactions. Furthermore, they create a generic and parameterized structure, which supports transferability to platforms without re-implementation.

1.5.1.3 Incremental bottom-up application of formal methods, ensuring effective testing and guaranteed system characteristics

The methodology developed in this thesis is used to implement software for experimental aircraft. Therefore, a deterministic automation with guaranteed system properties is required. An incremental bottom-up approach with formal methods is developed to archive this objective. At each implementation level, formal methods are used to guarantee certain output conditions under assumed input constraints. This enables the efficient use of formal methods since the scope is limited to each subsystem. The approach is then applied incrementally from the lowest to the highest level, which creates a seamless requirement coverage.

1.5.2 Flight Control System Automation

The FCSA is devised from the objectives presented in Subsection 1.4.2. The developed contributions, beyond the current state of the art, are introduced in the following and explained in more detail in Chapter 4.

1.5.2.1 Strategy for switchability between various modes on different authority levels, enabling experimental automation

To create a basis for automatic operation and to enable the overall automation, a strategy for the switchability between various operating modes on different control authority levels is developed. The FCSA needs to access all hierarchy levels within the cascaded control loops, therefore entry points between each adjacent control loop need to be defined. The position of all switches in those entry points is consistently managed by the automation to create the required overall system response and to ensure the consistency of the entire state machine.

1.5.2.2 Operational management concept for multi-user experimental OPVs and UAVs, increasing mode awareness

To increase the mode awareness and reduce the probability of mode confusion during the operation of experimental UAVs and OPVs an operator-centric operational concept is developed, which consists of multiple principles. The operator has the responsibility and therefore also needs command authority. Additionally, the operator needs to be actively involved in all major transitions. To support this the automation needs to be deterministic as well as predictable and provide enough information to the GCS.

To fulfill the objective of multi-user access an operational mode management strategy is developed granting accessibility to all hierarchical control levels. This includes mode resource management and mode compatibility management.

1.5.2.3 Automatic operational and malfunction contingency procedures for continuous operation in non-nominal circumstances

Automatic contingency procedures are integrated within the FCSA to allow for robust automation and continued fully automatic flight even in non-nominal circumstances. Those are divided into two groups of procedures. Operational contingency procedures are intended for a quick reaction to changes in the mission objectives, while malfunction contingency procedures allow for continued automatic operation even in case of sensor failures. Examples include modes that will fly the aircraft back to its home base, letting it hold at the current position, and modes that degrade to alternatives if necessary resources are no longer available.

1.5.3 Flight Test Maneuver Injection

The FTMI is devised from the objectives presented in Subsection 1.4.3. The developed contributions, beyond the current state of the art, are introduced in the following and explained in more detail in Chapter 5.

1.5.3.1 Generic design pattern, providing a free maneuver parametrization without reimplementation

A generic and maneuver-independent design pattern is developed that provides a free maneuver parametrization without re-implementation. This modularized systematic leads to a clear and explicit visibility of action alternatives. Maneuver generation modules are only implemented in one location, which creates distinct boundaries between multiple developers and allows for an easy extendibility in the future. This novel approach uses a generic parametrization and selection of maneuvers for maximum flexibility.

Additionally, a distributed design is developed to allow for the future separation of the injection onto different computers. This ensures that the contradictory requirements of large memory space and real-time capabilities can be met simultaneously.

1.5.3.2 Dynamic flexible choice of injection points on multiple control levels, enabling a generic implementation and safe execution

A flexible choice of injection points on multiple control levels of the cascaded control loop is created using a dynamic allocation matrix. The matrix assigns the maneuver signal to the control surfaces or command inputs for the flight control loops, based on a parameter set. This enables a generic implementation for multiple aerial platforms that can be tested completely, which results in a safe execution. Additionally, this dynamic assignment creates the framework to enable multi-axes execution, superposition for trim offsets as well as additional support functions like wing-leveling.

1.5.3.3 Individual trim point verification and automatic trim point capture for safe and effective flight testing

Advanced features are developed to support safe and effective flight testing. The individual trim point, which is defined for each maneuver, is used to perform an individual trim point verification before automatically starting the maneuver. This increases safety because maneuvers can only be executed in intended environmental conditions, while at the same time increasing effectiveness because the variation of the initial conditions is reduced. Novel features that further increase the effectiveness, utilize the flight control system to automatically capture the trim point, while constantly verifying it and automatically activating the respective maneuver if the trim point is confirmed.

1.6 Outline

The motivation for this thesis and background is presented at the beginning of this chapter. The current state of the art for each part is discussed in the following in Section 1.3. The derived objectives are presented in Section 1.4 before the developed contributions are described in Section 1.5. They show the approach on how to eliminate the shortcomings of current solutions and additional features developed in this thesis. In the following, the remaining chapters of this thesis are outlined.

In Chapter 2 of this thesis, the demonstration platforms used to prove the real-life applicability of the methodology and functions presented in this thesis are described in detail. Their respective project scope, operational scenario, and the range of demonstrated functions are illustrated to provide a better understanding of the development context. However, even more, detailed information about their hardware and software architecture is given in the respective chapter, where the aircraft is used as a demonstration platform.

The methodology, which is developed to design and implement the automation functions presented in this thesis, is described in Chapter 3. It's comprised of the theoretical background of automata and steps to design higher-level automation. Additionally, the implementation is introduced and various testing and verification methods are discussed.

The *Operator-Centric Multi-User Flight Control System Automation for experimental UAVs and OPVs with Contingency Procedures (FCSA)*, presented in Chapter 4, is capable of administering a cascaded control loop with respect to multiple users or automatically based on sensor information and integrity. In this chapter, the system architecture of the respective demonstration platforms, the various operating modes, and the transition conditions and actions from one mode to another are explained. Furthermore, the *Loiter Automation (LA)* is presented as one of the examples for contingency maneuvers within the FCSA, and the command injection points are discussed in detail before flight tests are presented. Those real-life applications of the developed software, prove the applicability to multiple platforms before a short summary is concluding the chapter.

In Chapter 5 the design and development of the *Multi-Maneuver Multi-Control-Level Flight Test Maneuver Injection with automatic Trim Point Capture and Verification (FTMI)* are presented. It is able to generate a variety of signals into various parts of the control loop and also directly to the control surface actuators. The chapter includes the system architecture as well as the allocation matrix and injection points of the developed software. Additionally, the operation modes and the transition conditions and actions of the main state machine are presented. Furthermore, the available maneuvers are shown before real-life flight tests prove the functionality of the software within the projects. The chapter is concluded with a short summary of the archived contributions.

Finally, a conclusion of the developed contributions summarizes this thesis with respect to the Methodology for System Automation, the Flight Control System Automation, and the Flight Test Maneuver Injection. Additionally, an outline of future works in those fields is given, which ends this thesis.

Chapter 2

Aerial Demonstration Platforms

The methodology and algorithms presented in this thesis are used on various aerial demonstration platforms and in different operational scenarios. Since flexibility and portability is an important design requirement, this chapter provides an overview of the demonstration platforms. This includes the project objectives and the resulting operational profile of the aircraft. The profile is comprised of the type of aircraft, number and types of operators, the number of data links, and a representative mission objective. Additionally, the role of the developed functions in the overall concept of the aircraft is presented and project and/or aircraft-specific challenges are highlighted. An overview of these points and the applicable function of each platform is given in Table 2.1. As mentioned before, all Optionally-Piloted Vehicles (OPVs) were only used with an onboard Safety Pilot (SP). Furthermore, a Flight Operator (FO) and/or an External Pilot (EP) can be used.

While the developed functions are not limited to a specific platform, within the scope of this thesis, there are demonstrated on a subset of the available demonstration platforms. The *Operator-Centric Multi-User Flight Control System Automation for experimental UAVs and OPVs with Contingency Procedures (FCSA)* is developed for use in Unmanned Aerial Vehicles (UAVs) as well as in OPVs. Consequently, it is tested on *SAGITTA* (UAV) and on the *DA 42* (OPV). The *Multi-Maneuver Multi-Control-Level Flight Test Maneuver Injection with automatic Trim Point Capture and Verification (FTMI)* was used on *ELIAS* and the *Do 228* to improve the quality of the Flight Dynamic Model (FDM) and to investigate actuator accuracy and increase controller performance.

Table 2.1: *Platform Overview*

		Type	Operator			Data Links	Function	
			SP	FO	EP		FCSA	FTMI
Platform	SAGITTA	UAV		X	X	3	X	
	DA 42	OPV	X	X	X	1	X	
	ELIAS	OPV	X	X		1		X
	Do 228	Manned	X			0		X

2.1 SAGITTA

The *SAGITTA Research Demonstrator* is part of the "Open Innovation" initiative of the aircraft manufacturer Airbus [MPH2017]. The aim of the project was to develop new technologies for UAVs, for which *SAGITTA* acts as a test platform. Figure 2.1 shows *SAGITTA* on the apron during the first flight campaign in 2017.

SAGITTA is an unmanned downscaled demonstrator with a maximum takeoff weight of $150kg$. The aircraft is operated as a research prototype and has a wingspan of $3m$ and a length of $3m$. It is powered by two integrated turbines, delivering a total thrust of $600N$. The diamond-shaped aircraft features a novel eight-trailing-edge-flap design. The inboard flaps are used as elevators, the ones in the middle as ailerons, and the outer split-flaps as yaw controls. Its specifications are summarized in Table 2.2.

During the missions, *SAGITTA* has three operators, one FO and two EPs. The FO is located in a Ground Control Station (GCS) and responsible for the overall mission and switching between higher-level automation modes like automatic takeoff and waypoint-based flight. The two EPs are located close to the runway and act as a backup for the FO. They can take over control in case of a failure of one of the higher-level automation modes of the Flight Control Computer (FCC). In that case, the FCC reverts to a lower level mode, which requires less sensor information.

Simulations preceding the first flight have shown that recognizing the attitude of the aircraft, due to its speed, size, and shape was very difficult. Additionally, due to its shape, complete manual control is nearly impossible for a pilot, which resulted in concerns when proposing the EPs as a backup.

Figure 2.1: *SAGITTA at Air Force Base Overberg (FAOB), South Africa, in 2017*

Table 2.2: *SAGITTA Specifications (adapted from [Air2017])*

Wingspan	3m
Maximum Take-Off Mass	150kg
Length	3m
Propulsion	2 x 300N turbines
Demonstrator Scale	1:4
Class	Research Prototype

Due to the, above-mentioned, concerns regarding the EPs as a backup, it was decided to perform a fully automatic maiden flight, which includes automatic takeoff and landing. The EPs should only intervene if a loss of the aircraft seemed inevitable. Therefore, the Flight Control System (FCS), including the FCSA, could only be tested in simulation and had to work without errors, when testing it in flight for the first time. That made the software of the FCC more critical, which in consequence led to even more testing and verification of the automation and control loops.

The command and control data link architecture is based on a primary Flight Data Link (FDL) and an additional Mission Data Link (MDL). The FDL is only used for sending critical command and control data to the aircraft and return sensor data, which is important for the FO. The MDL is primarily used for mission-level data, like video and images, but all data from the FDL is mirrored on the MDL for redundancy. An additional Termination Data Link (TDL) was used to create an independent possibility to abort the mission at any time.

Due to the contrasting level of involvement of the FO and the EP, the FCSA has different modes for both of them. They are separated into ground and airborne modes since different controllers are used in the two cases. The modes for the FO include automatic takeoff and landing, GPS-based waypoint control, and flight based on autopilot commands like altitude, heading, and speed.

Additionally, to those nominal operating modes, there are several contingency modes for both, operational changes and malfunctions. Because of the unknown reliability of the data links and the Global Positioning System (GPS) reception, contingency procedures for them are included within the FCSA. If a fault is detected, specific modes cannot be activated anymore, lower level modes are automatically activated if necessary or the FCSA will start automatic procedures that are designed to mitigate the malfunction.

SAGITTA successfully completed its fully automatic maiden flight in July, 2017, [Zim2017]. During that flight the FCS controlled the aircraft and the FCSA administered the cascaded control loop and switched between different modes as commanded by the FO. The second flight was conducted shortly after the first one and included more sophisticated modes of the FCSA. Further flight tests had to be canceled due to the time and budget restrictions of the project.

2.2 DA 42

The *DA 42* MNG (**M**ulti-Purpose Platform **N**ew **G**eneration) by Diamond Aircraft is owned by the Institute of Flight System Dynamics (FSD) of the Technical University Munich (TUM). It is the result of the development of a versatile research aircraft incorporating Fly-by-Wire (FBW) started in 2008 and also known as *DA42 MNG FBW Research Aircraft*. It has been used as an experimental platform in various projects and campaigns. The aircraft with the call sign `OE-FSD` is displayed in Figure 2.2.

The optionally-piloted *DA 42* is the multi-purpose version of the upgraded DA42, which is a Part 23 Class II aircraft with a maximum take-off mass (MTOM) below $2000kg$. It was equipped with a versatile flight test instrumentation, a multi-stage safety system, and an experimental FBW system [Pet2017]. Therefore the FCS can access all axes of control: elevator, aileron, rudder, and throttle levers. The general size, weight, and power specifications are summarized in Table 2.3.

The flight crew of the *DA 42* consists of three persons. The SP, in the front left seat, controls the aircraft whenever the FCS is deactivated and can also take over control in case of a malfunction. The test pilot, in the front right seat, can use a (passive or active) control stick for the pilot in the loop tests. The third crew member is the flight test engineer, in the left rear seat. Various control and monitoring equipment for the flight test instrumentation, safety system, and FBW system is located in the space of the removed rear right seat, so they are easily accessible for the flight test engineer.

Figure 2.2: *DA 42 at Wiener Neustadt East Airport (LOAN), Austria, in 2017*

Table 2.3: *DA 42 Specifications (adapted from [Dia2018])*

Wingspan	13.55m
Maximum Take-Off Mass	1999kg
Length	8.56m
Height	2.49m
Propulsion	2 x Austro Engine AE300 (123.5kW)
Class	Part 23, Class II

The test pilot can use an experimental Mode Control Panel (MCP) for the higher-level automation modes. The control sticks and MCP are designed in a way so that they can be used onboard the aircraft or within the GCS utilizing the data link. Depending on whether the FO or EP uses them onboard the aircraft or on the ground, the control laws are adapted to account for the different scenarios.

The *DA 42* is equipped with one data link, which is used if control and/or monitoring from the ground is required. The data link is used to transfer command and control data as well as necessary sensor information for monitoring purposes. Due to the safety pilot onboard additional data links are not required.

The structure of the FCSA is designed in a way to support the FO and EP of the *DA 42*. In the case of being used as an OPV both, the FO and EP can be onboard the aircraft or within the GCS. If the EP is onboard the aircraft, a direct-law (directly mapping control stick deflections to surface deflections) and other control modes can be used. If the EP is controlling the aircraft via data link special control laws that account for larger transmission delays with additional safety features are used.

The contingency procedures for the OPV are focused on dealing with operational changes while allowing for continuous automatic operation. Those include overriding altitude and speed commands, while laterally following a GPS-based waypoint trajectory. They can be used to avoid unexpected obstacles or to adjust the timing of the mission. Additionally, a loiter pattern can be initiated while flying in trajectory or autopilot control mode. Depending on the previous mode different loiter maneuvers are activated to allow for seamless continuation of the mission. Furthermore, modes like return-to-base can be used to abort the mission and return to the home base. The last two, loiter and return-to-base, even have sophisticated extensions to deal with GPS loss.

The first flight using the FCS system, including the FCSA, was successfully performed in January, 2016. Since then the FCS in the *DA 42* has been used on various flight tests and demonstration missions. This even includes fully automatic flight including automatic takeoff, GPS-based mission segments, and automatic landing without any intervention from the SP.

2.3 ELIAS

The Elektra One, which flew for the first time in March, 2011, is the first electrically powered ultralight aircraft in Germany and was developed by PC-Aero [Ele2018a]. The most recent version, the Elektra One Solar, is equipped with solar cells for improved flight duration [Ele2018b]. *ELIAS* (Electrical Aircraft IABG Acentiss) is a modified version, which has an electrical retractable landing gear for improved performance and an unobstructed view for cameras and sensors [Sig2016]. The aircraft, with the call sign `D-MELQ`, is shown in Figure 2.3 and is used for two projects at the FSD. The aim of project EUROPAS is to develop and integrate technologies for the first-time demonstration of using an electrically powered aircraft in the ultralight weight-class as an unmanned sensor platform [Lud2019b]. The subsequent project AURAIS is targeted at the development and demonstration of critical technologies concerning certification [Lud2019a].

ELIAS has been modified in several ways to be used as an optionally-piloted test platform. It has been equipped with actuators, including electromagnetic clutches and slip-clutches, for the three main control surfaces (aileron, elevator, and rudder). An additional actuator is used to move the throttle lever to enable automatic thrust control as well. Additional systems include, but are not limited to, the FCS, a Flight Management System (FMS), an Actuator Clutch Box (ACB), and a data link. Secondary surfaces and equipment like flaps, elevator-trim, and the landing gear cannot be controlled directly by the FCC. Desired positions for those are, besides other information, displayed on a specially designed mission display in the cockpit. The main specifications of the modified aircraft are presented in Table 2.4.

Figure 2.3: *ELIAS at Landshut Airport (EDML) in 2016*

Table 2.4: *ELIAS Specifications (adapted from [Ace2016])*

Wingspan	11m
Maximum Take-Off Mass	320kg
Propulsion	Electric Brushless Motor (20kW)
Maximum speed	150km/h
Cruise speed	110km/h
Maximum range	>150km
Maximum flight time	1.5h
Seats	1
Class	Ultralight

Because there is only one seat, the flight crew of *ELIAS* consists only of the SP, which can activate and deactivate the automatic FCS, monitor its performance, and execute necessary changes of secondary surfaces and equipment. Additional personal is located on the ground for coordinating the test and commanding and monitoring the FCS. At least two persons are needed, an SP onboard the aircraft and a FO in the GCS. The operation, in the class of ultralight aircraft, is performed according to national rules.

The hardware architecture of the research platform *ELIAS* includes a GCS and one data link. It is used to change modes of the FCS, upload waypoints, change commands for the autopilot, and monitor the system's response.

The aircraft is manually controlled by the pilot during takeoff and landing. Upon reaching a safe altitude the activation of the FCS can be requested by the operator in the GCS. If the pilot acknowledges activating the system, it can be confirmed using a button on the control stick. The Flight Control Clutch Automation (FCCA) [KH2017a], also developed by the author, within the FCC is activated and checks for further necessary activation conditions. It then synchronizes the actuators with the current surface positions and sends a command to the ACB to close the electromagnetic clutches. If the closing of the clutches is confirmed, within a certain time, the remaining modules of the FCS are activated and take over control of the aircraft. The most important characteristic of an experimental FCS within a manned aircraft is the possibility to safely disconnect it at any time. Within *ELIAS* there are several circumstances when the system will disengage itself and furthermore various ways the system can be disengaged by the SP.

The first automatic flight was successfully executed in November, 2016. Even though the FCS was able to fly the aircraft, analyzing the data revealed performance and accuracy issues, which were largely founded in the primary actuators and their connection to the control surfaces. In the following test campaigns the features of the FTMI, presented in this thesis, were utilized extensively to perform automatic actuator and flight tests. With the results, from those tests, the accuracy of the FDM and performance of the controller were improved to allow for satisfying automatic flight.

2.4 Do 228

The *Do 228-101* by Dornier is a universal platform, which is used by the German Aerospace Center / Deutsches Zentrum für Luft- und Raumfahrt (DLR) and has been in use for over 30 years for various research projects [DLR2019]. In a cooperation project with DLR and FSD this aircraft with the call sign D-CODE, as shown in Figure 2.4, is used as a development platform for an experimental three-axis autopilot.

This manned test platform, which is operated as a Part 23 Class IV commuter category aircraft, has been enhanced to create the possibility for automatic flight. It is equipped with, among others, a Safety Relay Box (SRB), an FCC, and actuators with smart Actuator Control Electronics (ACEs) including clutches for the main control surfaces. Since the FCS doesn't have access to the thrust levers, a Thrust Director (TDR) is used, showing the pilots the desired power setting. The aircraft provides workstations for research purposes for up to nine engineers, which are located in the back of the aircraft. Specifications for this uniquely equipped *Do 228* are summarized in Table 2.5.

The flight crew consists of at least two pilots. The safety pilot is located in the front left seat and responsible for the overall mission. The test pilot is located in the front right and can access a combined MCP and monitoring display. This MCP is the interface to the FCS, which consists of an autopilot and an inner loop, developed by FSD. The autopilot can either be used in *Selected Flight*, where the commands are entered through the MCP, or in *Managed Flight*, where the commands are provided through an interface.

Figure 2.4: *Do 228 at Oberpfaffenhofen Airport (EDMO) in 2021*

Table 2.5: *Do 228 Specifications (adapted from [DLR2018])*

Wingspan	16.97m
Maximum Take-Off Mass	5980kg
Length	15.03m
Height	4.86m
Propulsion	2 x Garret Turbopro TPE 331-5 (529kW)
Seats	15 (nine seats for research purposes)
Class	Part 23, Class IV

One or more additional flight test engineers can be placed in the rear. That makes it possible to analyze data while still in the air and adjust further tests as necessary, therefore allowing for very efficient testing, especially with the FTMI. Since space onboard the aircraft is sufficient for multiple flight test engineers, no data link is used in the research projects with FSD.

Due to the test pilot being onboard the aircraft, functions can be partially tested and improved before the FCS is fully activated. The FTMI, presented in this thesis, has been used to a great extent on the *Do 228* to perform automatic identification maneuvers and flight tests. The various maneuvers that can be injected on different levels of the cascaded control loop can be used for different analyses and applications.

The FCS of the *Do 228* has no trajectory generation or trajectory control within the software developed by FSD. Therefore the FTMI uses all levels of the control loop. At the lowest level, maneuvers can be used to directly control the actuators. They include aircraft parameter identification maneuvers like *Multi-Step* or actuator bandwidth identification maneuvers like *Sweep*s. On the next level, the maneuvers are injected as commands for the inner loop. The commands can be load-factor commands for the vertical motion or bank-angle commands for the lateral motion. Those maneuvers could be *Multi-Sine* or *Spline* sequences to evaluate controller performance. During those automatic flight tests, low-level autopilot functions, like yaw damping or holding the wings level, can be used to stabilize the aircraft, if those axes are not used during the test.

The first automatic flight, using the FCS developed by FSD, was performed in August, 2016. During the following flight tests, problems were discovered which were founded in the transfer function from the actuator to the control surface, which in turn were founded in unknown and therefore unaccounted transmission effects like loose fits, backlash, and hysteresis. To generate an accurate actuator model, which takes those effects into account, many automatic actuator tests were performed using the FTMI. After solving those problems the FTMI was used in multiple flight tests starting from October, 2019. During this campaign, the lateral dynamics of the aircraft could be identified more accurately, which was used in a gain tuning process to improve the lateral tracking performance of the inner loop.

Chapter 3

Methodology for System Automation

The developed *Design, Implementation, and Testing Methodology for System Automation - using State Machines in Stateflow (MTSA)* is presented in this chapter.

The MTSA is a generic methodology for higher-level system automation which includes design, implementation, testing, and verification. It enables deterministic, predictable, and robust automation of safety-critical software with guaranteed properties using state machines, which utilize the functionality of *MATLAB*, *Simulink*, and *Stateflow*.

The methodology approach for automation developed in this thesis differs substantially from the ones used in comparable systems and even from the ones used in large Unmanned Aerial Vehicles (UAVs) or manned aircraft. It is target at maintainability and expandability in an experimental context.

Since the methodology is the basis for automation functions used in real-life applications on experimental UAVs and Optionally-Piloted Vehicles (OPVs), safe and robust design has the highest priority in this development context. At the Institute of Flight System Dynamics (FSD) *MATLAB*, *Simulink*, and *Stateflow* are used in a model-based development process for aviation automation software. Therefore, the methodology must utilize those tools and must be applicable within the process.

The features of the methodology consist of a decomposition design strategy, modeling guidelines for implementation, and testing methods utilizing formal methods. Due to the encapsulation, changes only affect a small part of the model. Together with the modeling guidelines, this enables an incremental build procedure. The testing methods are very efficient since one level can be tested at a time and changes on one level don't influence already conducted tests on other levels.

In the following, the current state of the art, the resulting motivation from deficits, the objectives, and developed contributions are summarized. Afterward, the theoretical basics for this methodology are introduced. Subsequent sections present the developed methodology with respect to design, implementation, and testing.

Motivation - Methodology for System Automation

- A design and implementation approach for the development environment is necessary to reduce the complexity of higher-level automation
- Specific constraints must be met to suit the smaller development team, shorter project duration, future extension, and re-use
- Testing systematic based on formal methods is required to ensure safe and robust automation during experimental operation of UAVs and OPVs

The current state of the art with respect to the methodology part of this thesis is introduced in Subsection 1.3.1. The resulting motivation, due to the identified deficits, is recapitulated here. To reduce the complexity of higher-level automation finite state machines, a customized design and implementation approach is necessary. The development context, in particular, the smaller development team and shorter project duration require specific conditions to meet the required safety standard. Furthermore, a systematic testing method based on formal methods is not available with current solutions but is needed to ensure safe and robust automation during experimental operation of UAVs and OPVs.

The derived objectives, from the deficits in the current state of the art, are presented in Subsection 1.4.1 and summarized in the following. The overall objective that is presented in this chapter is the development of a methodology that enables the implementation of advanced higher-level automation finite state machines in *MATLAB*, *Simulink*, and *Stateflow*. Additionally, the methodology must ensure the manageability of the system during design and implementation. This is especially important because of the development context and should incorporate future extensions and usage on other platforms. The implemented system must be deterministic to ensure a safe and predictable behavior. Furthermore, the complete testability shall support robust automation with guaranteed properties.

Objectives - Methodology for System Automation

- Develop a methodology to implement advanced higher-level automation finite state machines in *MATLAB*, *Simulink*, and *Stateflow*
- Ensure the manageability of the system during design and implementation, focusing on future extension and usage on other platforms
- Implement a deterministic system that ensures a safe and predictable behavior as well as complete testability for robust and guaranteed properties

Contributions - Methodology for System Automation

- C1.1 - Hierarchical decomposition design strategy, minimizing complexity and optimizing testability
- C1.2 - Modeling guidelines for implementation, minimizing opacity and maximizing software maintainability
- C1.3 - Incremental bottom-up application of formal methods, ensuring effective testing and guaranteed system characteristics

The contributions of the methodology, presented in this chapter, are introduced in Subsection 1.5.1. The assignment of the individual contributions to the sections of this chapter is presented in the following together with the general outline.

At the beginning of this chapter, Section 3.1 describes the *Theoretical Basics*, which includes the background of state machines, fundamental automation theories, principles of modeling, and the introduction of the two most common state machines used today.

Afterward, Section 3.2 describes the developed *Design* process used in this methodology. It deals with common challenges in automation and how they are approached in this methodology. Next, the various design steps, that lead to the proposed safe and robust automation are introduced. Additionally, the decision logic and the hierarchical decomposition strategy are addressed. This section is associated with contribution *C1.1*.

In the following, Section 3.3, called *Implementation*, focuses on the development environment in terms of utilized software and is associated with contribution *C1.2*. It introduces the software used for building the state machines and their basic elements including an example. Further subsections define the level structure, which is used to lower the complexity and show how parameterization is used to adapt to different platforms, without changing the internal structure. Furthermore, certain design decisions and modeling guidelines, which were developed while improving this methodology, are presented.

The next section of this chapter, Section 3.4, takes a closer look at *Testing and Verification* and the testing process used and developed with this methodology. It introduces the multiple testing methods, and how they are leveraged to test different systems in various environments. Furthermore, the application of formal methods is presented and how they are used to design automata with guaranteed properties. This part of the chapter is associated with contribution *C1.3*.

The chapter is concluded with a short *Summary* and recapitulation, in Section 3.5, of the content and contributions made.

The basic idea for this MTSA and parts of it have been previously published [KH2018a]. However, this chapter takes a more detailed approach and includes various undisclosed aspects of the methodology.

3.1 Theoretical Basics

This section introduces the theoretical basics of state machines. This includes the history of state machines beginning in the early 20th-century, automata theory including various types that can be separated by their computational or representational power, different possibilities for modeling which are used in this thesis, and the two most common types of state machines being *Mealy* and *Moore* finite state machines.

The History of state machines, Subsection 3.1.1, provides an overview of the history and development of state machines. It starts with the studying, analysis, and synthesis of relay switching circuits in the early 20th-century, through the crucial publications of *George H. Mealy* and *Edward F. Moore* in the 1950s in which the basis for the two most common state machines of today was developed, all the way to their modern descendants, where multiple interacting nodes are connected through a network. This overview is given, to better determine the current state of development in the field of state machines.

Automata Theory, Subsection 3.1.2, introduces different classes of automata, which are established based on their computational or representational power. On the first level of separation, they include Combinational Circuits, Sequential Machines, Finite State Automata, Pushdown Automata, and Turing Machines. Additionally, Finite State Automata is divided, based on their output, into Acceptors, Classifiers, and Transducers. Furthermore, Transducers can be divided even further depending on the information used to determine the output and next state into Sequencers, *Moore* machines, and *Mealy* machines. All of those are presented with an example and compared to each other in terms of complexity and performance.

State Machine Modeling, Subsection 3.1.3, presents different methods of modeling. They consist of the Functional View represented as Functional Program Code, the Imperative View represented as Imperative Program Code, and the Feedback View represented by the Tabular, Matrix, and Graphical Description. With those, a wide range of representations is outlined and the most relevant for this methodology are described in more detail. The Transition Matrix will be used throughout this thesis to give an overview of possible state transitions in a written form, while the Graphical Description represents the implementation in *Stateflow*.

Mealy and *Moore* finite state machines, Subsection 3.1.4, compares today's most commonly known types of state machines. The *Mealy* machine, invented by *George H. Mealy* in 1955, and the *Moore* machine, created by *Edward F. Moore* in 1956. The subsection presents their renowned publications and a comparison of the two machines. In this comparison, an example is used to highlight the differences between the two, including the respective transition table, input-output mapping of an example input, and the graphical representation.

3.1.1 History of State Machines

This subsection gives an overview of the history and origins of state machines but does not claim to be complete. The references can be used as a basis for more detailed information.

The history of state machines originates from the studying and analysis of switching circuits. In the early 20th-century, those circuits consisted of relays or vacuum tubes which were later replaced by transistors.

Today, the most commonly known types of finite state machines are the *Mealy* machine and the *Moore* machine. Those were developed by *George H. Mealy* and *Edward F. Moore* respectively at about the same time in the 1950s. During that time they both worked for *Bell Telephone Laboratories*, which was an industrial research and scientific development company. Their work is strongly related to the work of *Claude E. Shannon* and *David A. Huffman* who were both parts of the *Massachusetts Institute of Technology* during that period in time. It is even based on previous analysis and synthesis of relay switching circuits by *Claude E. Shannon*, *G. A. Montgomerie* and *David A. Huffman* conducted in the 1930s to 1950s.

In 1938 *Claude E. Shannon* performed a symbolic analysis of relay and switching circuits [Sha1938]. Complex electrical systems during that time required intricate interconnections of relay contacts and switches. Examples he stated for such circuits are automatic telephone exchanges and industrial motor-control equipment. He performed a mathematical analysis of certain properties of such networks with special attention to the problem of network synthesis. The requirement was to find a circuit incorporating certain characteristics and since the solution to such a problem is not unique, a method of finding the one circuit with the least amount of relay contacts was investigated.

G. A. Montgomerie presented an algebra of relay circuits in 1948 [Mon1948]. He described a system of symbols by which the elements of a relay circuit can be represented. Linking those symbols together then results in an algebraic expression that corresponds to the given circuit. Rules for these expressions are then developed which link the operation of the circuit to the behavior of the algebraic expressions. Using those expressions, methods are described to simplify complex circuits.

The synthesis of sequential switching circuits was developed by *David A. Huffman* in 1954 [Huf1954]. He established an orderly procedure to transform the requirements of a sequential switching circuit to the requirements of several combinational switching circuits. The difference between a sequential and combinational switching circuit is, that the sequential one has a memory where the combinational one doesn't have memory. Those and other classes of automata and their differences are described in Subsection 3.1.2. He introduced the flow table and the transition index. The flow table is a table that describes the requirements of a circuit and the transition index is a variable that indicates the stability of a switching device. He investigated the role of indirectly controlled switching devices and used the resulting philosophy for synthesis procedures.

The foundation of the *Mealy* machine known today was presented by *George H. Mealy* in his article "*A method for synthesizing sequential circuits*" in 1955 [Mea1955], which was part of *The Bell System Technical Journal*. He developed the theoretical basis of sequential circuit synthesis, which prefers formal procedures over intuitive ones. During that time two methods existed for reducing circuit requirements. The merging process described by Huffman on the flow table and the procedure by *Edward F. Moore*[1]. Merging is easier but does not result in a complete reduction at all times. Hence *Edward F. Moore* developed his additional procedure to ensure a full reduction in the remaining cases. *George H. Mealy* then found a solution, which is as simple as merging and results in a complete reduction more often. Additionally, he found a way to extend Huffman's method, which was designed for relay circuits, to be also used with switching circuits.

Edward F. Moore presented the foundation of the *Moore* machine known today in his article "*Gedanken-Experiments on Sequential Machines*" in 1956 [Moo1956]. This article is part of the book "Automata Studies" [SM1956], which presents different challenges and solutions to the field of automata. The article of *Edward F. Moore* is part of a chapter that deals with finite automata, which has a finite number of inputs, outputs, and internal states. Such "devices" can be described by two functions, one for the next state and one for the output, which can both be dependent on the current state and the current input. Many interesting problems arise from this rather simple description and essentially all physical machines and even a brain can be described like this [SM1956, p. vi]. *Edward F. Moore* investigated the abstract properties of sequential circuits by treating them as black boxes and only by manipulating the inputs and examining the outputs.

The two publications of *George H. Mealy* and *Edward F. Moore* created the basis for the use of state machines in many areas. While they introduced the concept for the two types of state machines they could not solve all related problems.

In 1971 *John H. Conway* expanded on Moore's concept by improving length bounds for his experiments and included examples like the Enigma machine, bombs, and detonators [Con1971]. He focused his work on finite and deterministic machines, arguing however that an input device only accessible to gremlins could be used to include probabilistic theory and therefore support non-deterministic machines [Con1971, p. 1].

Mike Holcombe examined various applications of the idea of *Mealy* machines in 1982 [Hol1982]. He developed a way to decompose machines in biology, biochemistry, and computer science and simulate them by simple ones using fundamental ideas from algebra. According to him, the mathematical theory of different types of machines has led to cybernetics, which has been immensely relevant in fundamental research.

The term cybernetics was introduced by *Norbert Wiener* in 1948 [Wie1948], which evolved to the modern term Cyber-Physical Systems (CPS) in the early 2000s [LS2017].

[1]Even though *Edward F. Moore's* procedure was officially published only one year after *George H. Mealy's* article, one can assume, and statements in his article confirm, that he had early access to the work of *Edward F. Moore*.

3.1.2 Automata Theory

Based on the computational or representational power of the system, automata theory can be divided into different types like Combinational Circuits, Sequential Machines, Finite State Automata, Pushdown Automata, and Turing Machines. Depending on the type of output the Finite State Automata can be further divided into Acceptors, Classifiers, and Transducers. The Transducers in turn can then be even further divided, depending on the information used to determine the output and next state, into Sequencers, *Moore* machines, and *Mealy* machines.

The nested overview of different types of automata is depicted in Figure 3.1, ranging from low (bottom) to high (top) performance[2]. A further subdivision of Pushdown Automata and Turing Machines would be possible but is omitted here since the main focus is on Finite State Automata. [Sim1999, p. 3]

After discussing some general attributes and definitions of automata, this subsection is introducing each type, of the ones mentioned above, in more detail.

A common distinction between automata is their determinism. Any actual machine inherently has some kind of unreliability and therefore non-deterministic behavior. However, due to the critical development environment of airborne software, only by design deterministic automata are discussed in this thesis.

Another distinction can be made, depending on whether or not the automata is finite. However, since all actual machines are somewhat limited and therefore finite, this distinction will be omitted here as well. Depending on the problem, the concept of non-finite automata can be useful, but in all real-life applications having automata with a sometimes (very) large but finite memory is sufficient to solve the problem.

As discussed in the previous subsection, the study of automata originated from the analyses of switching circuits, composed of logic gates. This resulted in automata theory and various concepts of state machines. Therefore, in the following, the words circuit, logic, machine, and automata are treated as basic synonyms and only used individually to distinguish their origin.

Another two commonly used attributes with automata are Context-Free language and Regular Expressions. They are often used in the context of finite state automata and Pushdown Automata and are ways to describe the behavior of state machines in an abbreviated form. More precisely, one would have to distinguish between Context-Free languages, which have a Context-Free grammar, and Regular languages, which have Regular Expressions. While Regular Expressions are recognized by both Pushdown Automata and Finite State Automata, Context-Free grammar is only accepted by Pushdown Automata, thus demonstrating, that Regular Expressions are a subset of Context-Free language. However, due to the fact that both are not relevant in the context of higher-level automation, they will not be discussed further.

[2]The order presented here, which is based on complexity and performance, does not reflect their time of invention or discovery in history.

In the following, different types of automata, which are relevant to this thesis are introduced while discussing their differences. This includes two types that are less and two types that are more powerful than finite state machines, thus providing a better understanding of the development environment and the allocation of finite state machines in general.

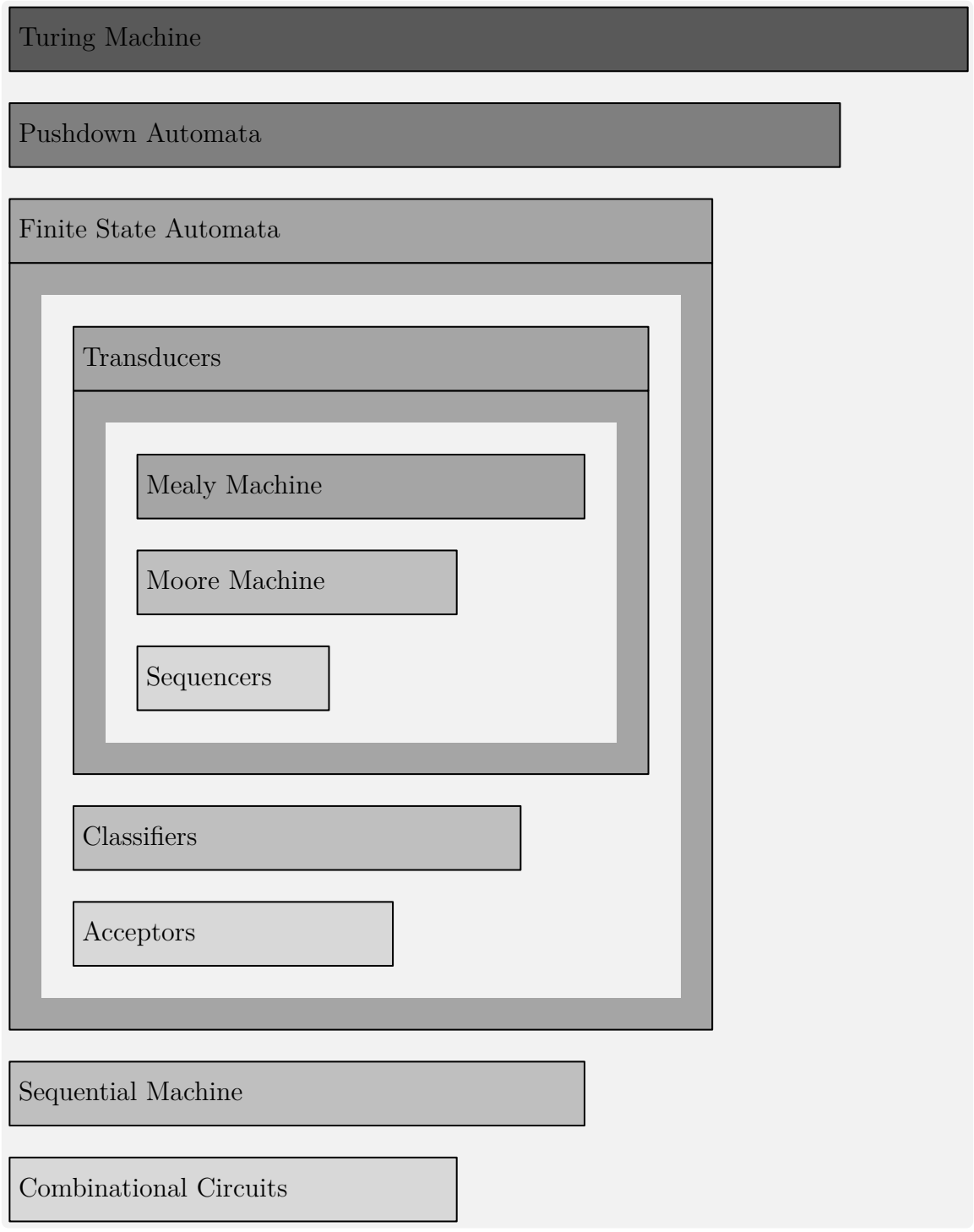


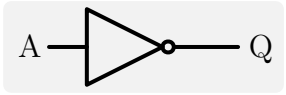
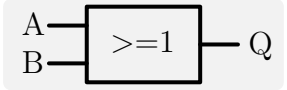
Figure 3.1: *Automata Theory (adapted and expanded from [Sim1999, p. 3])*

3.1.2.1 Combinational Circuits

Combinational logic is a term used in technical computer science and refers to a network of logical switching elements. Those elements also referred to as logical gates, can be represented using graphical symbols, described mathematically using Boolean algebra, and implemented using *ANSI C-Code* or Truth Tables. The graphical symbols are defined in ANSI/IEEE Std 91-1984 [IEE1984] and its supplement Std 91a-1991 [IEE1991, p. 60f]. Boolean algebra was introduced by *George Boole* in 1847 [Boo1847] and expanded in 1854 [Boo1854], but became only known as such in 1913 [Hun1933, p. 278].

The three elementary gates are the NOT gate, also referred to as the inverter, the AND gate, and the OR gate. The NOT gate has one input A and one output Q, while the AND and OR gate have an additional input, B. Their graphical representation, Boolean expression, *ANSI C-Code*, and Truth Table are depicted in Table 3.1. Two different graphical symbols are defined by the IEEE, the distinctive shape (upper image), and the rectangular shape (lower image). Corresponding Boolean Algebra is shown in the next row in two different notations and the resulting *ANSI C-Code* is shown in binary (upper row) as well as logical form (lower row) below that. The relationship between the input or inputs and the output is illustrated in the respective Truth Table. A Combinational Circuit can consist of an arbitrary number of logic gates and connections between those.

Table 3.1: *Elementary Logic Gates*

NOT	**AND**	**OR**
		
		
$\neg A$ $\overline{A}$	$A \wedge B$ $A \cdot B$	$A \vee B$ $A + B$
$\sim A$ $!\, A$	$A \,\&\, B$ $A \,\&\&\, B$	$A \mid B$ $A \mid\mid B$

A	**Q**
0	1
1	0

A	**B**	**Q**
0	0	0
1	0	0
0	1	0
1	1	1

A	**B**	**Q**
0	0	0
1	0	1
0	1	1
1	1	1

3.1.2.2 Sequential Machine

As discussed in the previous paragraph, combinational circuits are a connection of multiple elementary logic gates. Additional gates, like NAND, NOR, XOR, and XNOR, are constructed from the elementary ones and used as encapsulated subcircuits for better clarity and understanding. It is theoretical even possible to only use two elementary gates (NOT and AND or NOT and OR) to construct all others. However, no matter how many elements it contains, the Combinational Circuit is by definition time-independent, because the output values depend only on the present input and not on past values. Therefore, a certain input will result in an "immediate" output. Depending on the number of elements, it is obvious, that this description is not sustainable when implementing such a circuit in hardware, with transistors, due to their inherent output propagation delay.

The advanced property of a Sequential machine is, that the output depends not only on the current but also on past inputs. To determine the current output of the circuit it is, therefore, necessary to specify a sequence of inputs, which makes it time-dependent. To gain this advantage the machine needs to "remember" its past input values. This is also referred to as the "memory" of the machine. An input sequence leads to a certain condition of that memory, which is called the "state" of the machine.

The restriction of non-feedback connections in Combinational Circuits is omitted to create logic circuits with memory. Feedback, in this case, refers to a connection from an output to an input that is located before, in terms of signal flow, the output (e.g. an output of a logical gate is also used as one of its inputs). A basic design element to create a sequential circuit with memory is called "latch". This requires a feedback connection which forms a closed-loop. To ensure the correct and deterministic behavior of the circuit it is required to provide a timing signal to the memory elements to define how long to remain in a certain condition and when to sample new input data. Depending on whether or not the same signal is used for all elements or separate signals are defined, the machine is called a synchronous or an asynchronous sequential circuit. The timing signal, more commonly known as the "clock", is oscillating between two values with constant pulse lengths and spacing between two pulses. Memory elements that change their output based on the clock, rather than on a data input, are called "flip-flops". The change in output or change of state occurs when the clock transitions from zero to one (rising edge) or from one to zero (falling edge). [RK2004, p. 322ff]

There are various latches and flip-flops, but one of the most commonly known memory elements is the "S-R Latch" and the "S-R Flip-Flop". The "S-R Latch" has two inputs and two outputs and consists of two OR and two NOT gates. The inputs consist of S for setting and R for resetting the output Q. An additional output Q' is the complement of Q. The output is held at either `true` or `false`, depending on which input was last `true`. The "S-R Flip-Flop" consists of two "S-R Latches", four AND gates, and one NOT gate. It has one additional input for timing, CLK. The functionality remains the same, but the inputs are only sampled on the rising edge of the clock.

3.1.2.3 Finite State Automata

The following type of automata are based on the concept of sequential circuits. It is used to create an abstract representation known as finite state automata [Hen1968, p. 1]. They can be divided into Acceptors, Classifiers, and Transducers. Transducers can then be further divided into Sequencers, *Moore* machines, and *Mealy* machines.

Acceptors have the lowest capabilities, within Finite State Automata and are also known as recognizers or sequence detectors. They only produce a single binary output, also known as the accepting or rejecting state, an indication of whether or not the sequence of input symbols has been "correct". Acceptors can contain multiple transition states, but only one rejecting and one accepting state and they don't use actions except when transitioning to the latter two states. They can e.g. be used to verify if an input is consistent with a certain passphrase. [Kel2001, p. 480f]

Classifiers enhance the concept of Acceptors and omit the requirement of a binary output. Therefore, they can have more than two terminal states and return more than just binary information. However, they still don't use actions in transitions. An example is a machine that counts the number of occurrences of a special character in a string.

All types of Transducers use actions between all states to generate a continuous (in terms of not only terminal) output. The next state and output are based on the current input and/or current state.

Sequencers, which are also called generators, only have a single-letter input alphabet. Therefore, they are only able to produce the same output, since the input symbol is not changing. [HMU2007, p. 28f]

Moore machines drop the requirement of a single-letter input alphabet and use the current state and current input to determine the next state. The output, however, is only dependent on the current state.

The *Mealy* machine is an extended *Moore* machine, which selects its output based on the current state and current input. In general, every *Moore* machine can be converted into a *Mealy* machine and vice versa. However, using *Mealy* machines has several advantages, like normally fewer necessary internal states and faster reaction to inputs.

The theoretical basics of the last two machines are discussed in more detail in Subsection 3.1.4, while Subsubsection 3.3.2.2 will describe differences in implementation and why only one of them is used for implementing functions in this thesis.

One way to imagine the functionality of a finite state machine is as a black box that is fed with an input tape from the right and prints out an output tape on the left. The mental picture was presented by *Mike Holcombe* [Hol1982, p. 47] and is depicted in Figure 3.2. In this illustration, both the input and the output tape move from right to left. The state machine has an internal state S, reads input symbols u, and prints output symbols y.

The *Mealy* machine is more powerful and therefore a *Moore* machine can be seen as a restricted *Mealy* machine. Consequently, in the following, only the function of a *Mealy* machine is described and used as an example.

In the beginning, the state machine is in state S_1 and reads the first input symbol u_1. Based on the state transition function F, which depends on S_1 and u_1, the new state S_2 is calculated as $S_2 = F(S_1, u_1)$. At the same time, the first output y_1 is determined using the output function G, which also depends on S_1 and u_1 ($y_1 = G(S_1, u_1)$). Therefore, the state machine has printed out y_1, is now in state S_2 and will read the next input u_2.

Generally speaking, the next state is depending on the current state and current input ($S_{i+1} = F(S_i, u_i)$). At the same time, the output also depends on the current state and input ($y_i = G(S_i, u_i)$. While the first statement also applies to *Moore* machines, the latter has to be adapted, by omitting the dependency on u_i in the output function G.

In this example in Figure 3.2, the input tape contains n input symbols. When all of them have been read, the state machine has also written n output symbols, up to y_n, has gone through n transitions, and is in the state S_{n+1}.

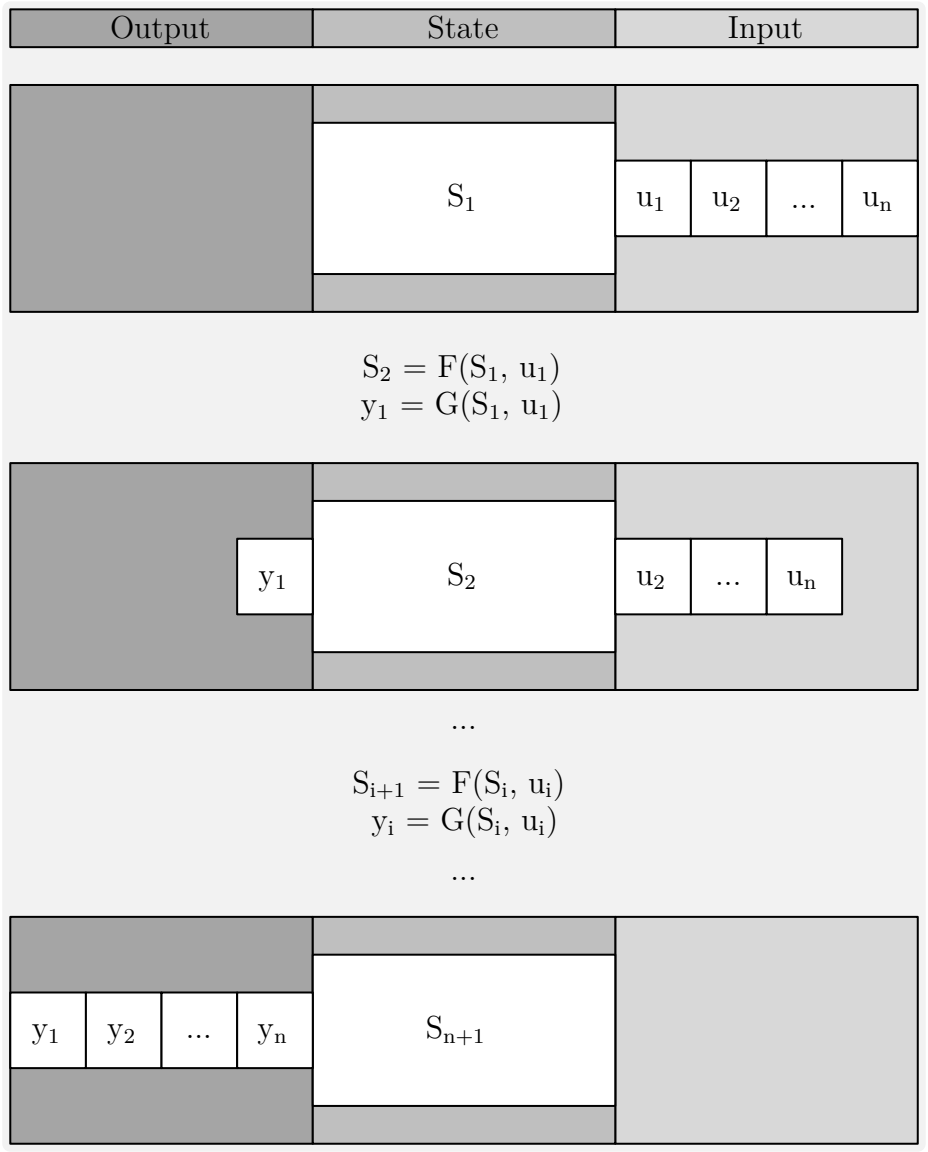


Figure 3.2: *Finite State Machine (adapted from [Hol1982, p. 47])*

3.1.2.4 Pushdown Automata

Despite the fact that the notion of pushdown tape has been used since 1954, Pushdown Automata (in this case Pushdown Acceptors) were first formalized in 1962 to 1963 by *N.Chomsky* and *R.J.Evey* [Gin1966, p. 81f].

The concept of Pushdown Automata introduces an enhanced Finite State Automata. Those machines are equipped with an additional tape, called the "stack". The stack acts like an additional memory space, where information can be stored and retrieved. An example is depicted in Figure 3.3.

As part of a transition, the Pushdown Automata can read and/or modify the stack. However, the scope for reading and modification is limited to the top element. Possible changes to the stack are: "push", "pop" or ignore. Writing a symbol to the top of the stack is referred to as "push". By adding another symbol, the rest of the stack can be regarded as being "pushed down", hence the name Pushdown Automata. Deleting the top element from the stack is also called "pop". If no modification is required the automation can also only read the top element or ignore the stack completely.

The stack can be used to determine the new state as well as to determine the output. Therefore, the dependency on the last stack element c_j is added to both, the state transition function as well as the output function ($S_{i+1} = F(S_i, u_i, c_j)$ and $y_i = G(S_i, u_i, c_j)$).

Furthermore, the Pushdown Automata includes a stack modification function H. It can be used to change the order of elements within the stack. However, further discussion is omitted here, and therefore it is also not shown in the schematic.

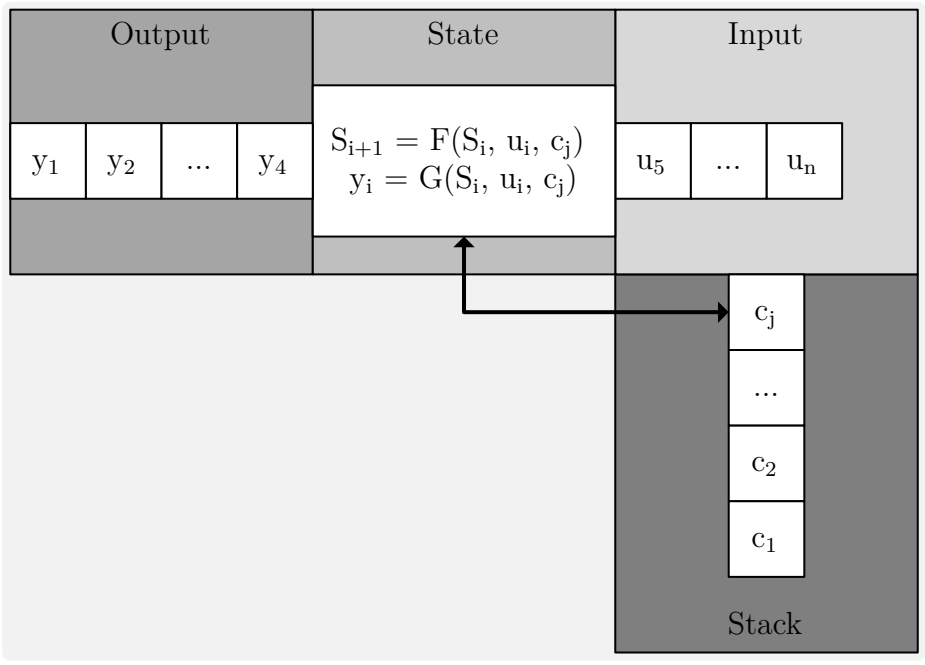


Figure 3.3: *Pushdown Automata*

3.1.2.5 Turing Machine

The concept of Turing Machines was developed almost 20 years before *George H. Mealy* and *Edward F. Moore* presented their work on finite state automata.

In his paper, written in 1936, "On Computable Numbers, with an Application to the Entscheidungsproblem" [Tur1936], *Alan M. Turing* presents a definition for a class of abstract machines. The concept of abstract machines, which became known as Turing Machines, also have a stack, like Pushdown Automata, but have one important advantage. Unlike Pushdown Automata, they have access to the complete stack making them more complex, but also more powerful. A visual representation of such a machine is shown in Figure 3.4.

With this extension, both, the state transition function, as well as the output function, can be dependent on an arbitrary element of the stack c_k leading to $S_{\mathrm{i}+1} = F(S_\mathrm{i}, a_\mathrm{i}, c_\mathrm{k})$ and $y_\mathrm{i} = G(S_\mathrm{i}, u_\mathrm{i}, c_\mathrm{k})$.

In contrast to Finite State Automata, the stack creates an unlimited and unrestricted memory for the Turing Machine. This makes the concept a much more accurate model of a general-purpose computer and enables the Turing Machine to do everything a real computer can do. [Sip2006, p. 139f]

The methodology and functions developed in this thesis are based on finite state machines. Nevertheless, due to the design steps presented in Subsection 3.2.2 together with internal (Subsection 3.2.3) and external (Subsection 3.2.4) decision logic they are enhanced to even higher-level automata. Depending on the specific functionality and implemented the complete automation can be considered a Pushdown Automation or Turing Machine.

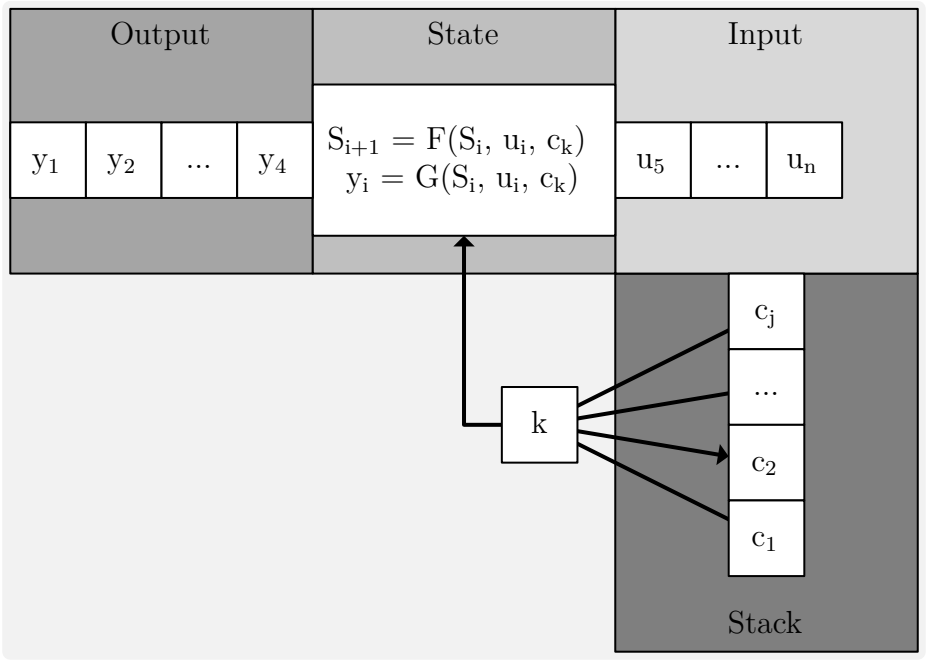


Figure 3.4: *Turing Machine*

3.1.3 State Machine Modeling

After introducing different types of automata in Subsection 3.1.2 this subsection gives an overview of different methods to model and implement automata. Those different notations are used to describe the behavior of finite state machines at different stages of the development and implementation process and throughout this thesis.

They include the Functional View represented as Functional Program Code and the Imperative View represented as Imperative Program Code. Additionally, the Feedback View includes the Tabular, Graphical, and Matrix Description. The general division of those notations is adapted from [Kel2001, p. 474ff] and listed in the following.

- Functional View
  - Functional Program Code
- Imperative View
  - Imperative Program Code
- Feedback View
  - Tabular Description
  - Matrix Description
  - Graphical Description

As an introduction to the different notations of finite state machines the example of an Edge-Detector is used throughout the following pages. An Edge-Detector is used to recognize a change (edge) within the input stream and report this with a given output value. In this case, the input stream consists of a finite number of integer variables that can either be "0" or "1". If a change is detected the output will be set to "1" for one cycle. In the following discussion, the output is defined to be "0" for the first cycle.

Depending on the step in the development process or the description within this thesis different notations are used. An example is modeled in each respective notation and their usage within this thesis is explained.

3.1.3.1 Functional View

The function of a state machine can be represented as a Functional Program. In this case, the behavior of the machine is modeled as a function from lists to lists. When one function has finished the processing of one input symbol it calls another function to process the remaining input, which is referred to as interrelation by mutual recursion. [Kel2001, p. 475f]

The Functional View with the Functional Program Code is listed for the sake of completeness only. It will not be discussed further, because it is not used within the development process of the methodology or applications presented in this thesis.

3.1.3.2 Imperative View

In the Imperative View, the state machine is represented using Imperative Program Code and the input and output are viewed as streams of integers. The function processing the current input and returning the respective output is presented as `detectEdge` in Listing 3.1. The wrapping function reading the current input, calling `detectEdge`, and writing the resulting output is omitted.

This view represented by the Imperative Program View is used for the notation of state machines when *ANSI C-Code* is automatically generated as part of the model-based development process in *MATLAB*, *Simulink*, and *Stateflow*. However, in this case, the *ANSI C-Code* is written by hand and not generated automatically from another representation within *MATLAB*.

This example represents the functionality of an edge detector as an imperative program. Advanced features like protecting the input from invalid data and saturating variables are omitted. Furthermore, the code is not optimized to show specific states but rather for easy understanding and representation of the functionality.

```
1  int detectEdge(int currentInput)
2  {
3    //local variables
4    static int state = -1;
5    int ret = 0;
6    //first execution, output always zero, save state
7    if (state == -1)
8    {
9      ret = 0;
10     state = currentInput;
11   }
12   //branch after first execution
13   else
14   {
15     //rising edge
16     if ((state == 0) && (currentInput == 1))
17     {
18       ret = 1;
19     }
20     //falling edge
21     if ((state == 1) && (currentInput == 0))
22     {
23       ret = 1;
24     }
25     state = currentInput;
26   }
27   return ret;
28 }
```

Listing 3.1: *Edge Detector - Imperative Program Code*

3.1.3.3 Feedback System View

The functionality of a state machine can be abstracted by using the state transition function F and output function G as introduced in Subsubsection 3.1.2.3. This is referred to as Feedback System View and is represented in Figure 3.5.

This notation of signal flow is not used explicitly in this thesis, but it is the basis for the Tabular, Graphical, and Matrix Descriptions, which are introduced in the following.

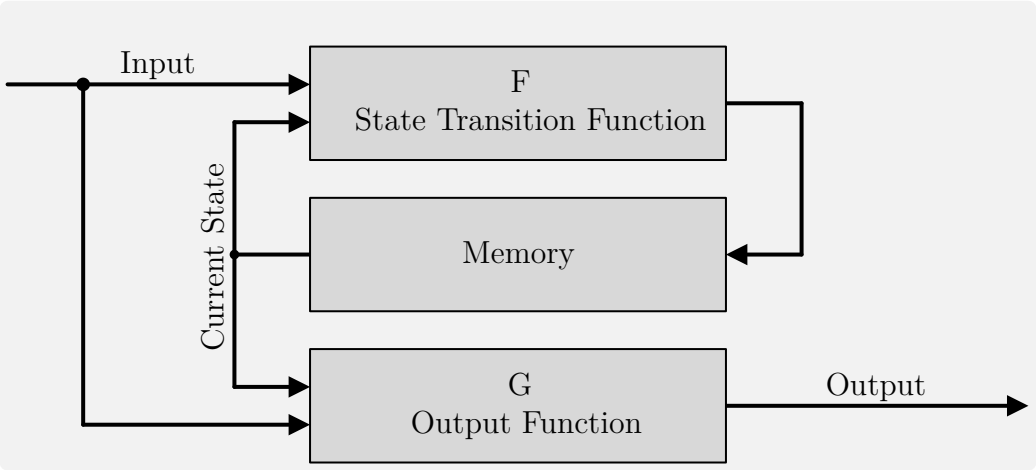


Figure 3.5: *Feedback View*

3.1.3.4 Tabular Description

The notation of the behavior of a state machine in a tabular description is referred to as a state-transition table. The respective notation for the Edge-Detector example is depicted in Table 3.2. The state-transition table is slightly different depending on whether the *Mealy* or *Moore* representation is used (c.f. Subsection 3.1.4), here the former is used.

Table 3.2: *Edge Detector - State-Transition Table*

Current State	Current Input	Output	Next State
init	0	0	S0
	1	0	S1
S0	0	0	S0
	1	1	S1E
S0E	0	0	S0
	1	1	S1E
S1	0	1	S0E
	1	0	S1
S1E	0	1	S0E
	1	0	S1

This table consists of four columns for the current state, current input, output, and next state. The initial state of the Edge-Detected is represented in the description as the current state, "init". However, this is not an actual state and only used to indicate the very first step. After that, the states $S0$, $S0E$, $S1$, and $S1E$ are used depending on the last and current input.

Transitions returning to the same state are not necessary for the Edge-Detector. However, they are included in this representation to highlight both input alternatives. As with the previous implementation of the Edge-Detector, an edge cannot be detected in the first step and the output is defined to be "0" after the first transition.

The Tabular Description is used in the following Subsection 3.1.4 to demonstrated the differences between *Mealy* machines and *Moore* machines. Additionally, it creates the basis for the Matrix Description.

3.1.3.5 Matrix Description

The Matrix Description which is also referred to as a two-dimensional state transition table is a compact combination of the Tabular and Graphical Description. The illustration for the example of the Edge-Detector is shown in Table 3.3. It shows departing states, including the additional "init" state on the left and respective arriving states on the top.

For the simple example of an Edge-Detector, the transition conditions and actions could be added directly in the matrix. However with the complex state machines, described in this thesis, this is not the case, and therefore only the available transitions themselves, without conditions and actions, are shown in a "from→to" notation.

As mentioned before, transitions returning to the same state are not necessary for the Edge-Detector and in general have more disadvantages than advantages. Within the matrix description, they are therefore always marked with a "-". Furthermore, functionally not available transitions are marked as "n/a" (not available).

This creates a general overview of available and unavailable transitions and will be used throughout this thesis to provide an easier understanding of currently discussed transition conditions or actions.

Table 3.3: *Edge Detector - Transition Matrix*

		To			
		S0	S0E	S1	S1E
From	init	init→S0	n/a	init→SS1	na
	S0	-	n/a	n/a	S0→S1E
	S0E	S0E→S0	-	n/a	S0E→S1E
	S1	S1→S0	S1→S0E	-	n/a
	S1E	n/a	S1E→S0E	S1E→S1	-

3.1.3.6 Graphical Description

The Graphical Description uses circles for the states and arrows for transitions. The circles include the name of the state and the arrows indicate possible transitions from one state to another in a u/y notation. If the correct input u is read the transition is executed and the output y is written. The graphical representation for the Edge-Detector is shown in Figure 3.6.

Self-recurring states are such that have a transition that changes an output but returns to its originating state. They were included in the tabular description to highlight both alternatives, marked with a "-" in the matrix description, and are also not used in the graphical representation. Furthermore, those states or transitions are not used within the methodology or applications presented in this thesis for reasons of clarity and complexity. They would lead to a graphical representation with sometimes fewer states but could, among other problems, lead to confusion on the implemented functionality.

This presented graphical notation of the behavior of finite state machines is the basis for the representation in *Stateflow*, where they have a slightly different representation, but a similar concept. The detailed introduction of state machines within the development environment is presented in Subsection 3.3.2.

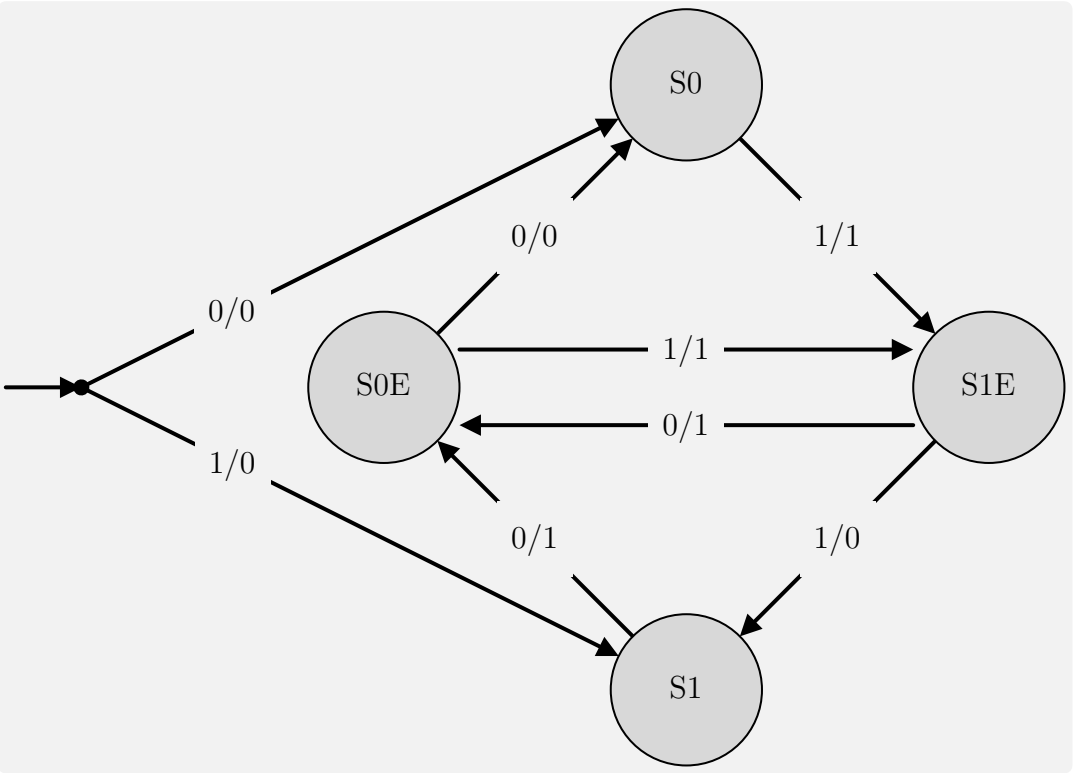


Figure 3.6: *Edge Detector - State Machine*

3.1.4 Mealy and Moore Finite State Machines

As stated before in Subsubsection 3.1.2.3, the most commonly used state machines are the *Mealy* and the *Moore* machine. Therefore this subsection describes them in more detail. The next pages include the key statement and an example machine from the original publications of *George H. Mealy* and *Gordon Earle Moore*.

Slightly adapted versions of their originally presented machines are shown in Figure 3.7 and Figure 3.8. In this example, they both have one input and one output, which can be either 0 or 1. Furthermore, they have four states (circles) in the same position. In this case, they have been renamed, $S1$ to $S4$, for better comparability. In the case of the *Mealy* machine, the transitions (arrows) are marked in a u/y notation, u being the necessary input and y being the resulting output. The *Moore* machine outputs depend only on the state and therefore the transitions only consist of the necessary input u, while the resulting output y is part of the state naming, $S_{\mathrm{i}}; y$. In both cases, multiple conditions are possible, which in the case of the *Mealy* machine are written in multiple rows, while the *Moore* machine distinguishes them by a comma.

The corresponding transition tables are depicted in Table 3.4 and Table 3.6. Due to different input-output timing, the columns of the tables are not in the same order and have different names. Notwithstanding the output timing, it can be seen, that the machines and their state change reaction to inputs are completely identical.

Actually, a closer look at the state diagrams reveals only one small, but very important, difference. In the *Mealy* machine, all transitions leading to $S2$ will result in an output of 0, while all transitions to $S3$ yield an output of 1. In contrast, when using the *Moore* machine, states $S2$ and $S3$ will issue the complementing output to the *Mealy* one.

In order to examine this behavior more closely, an arbitrary input with starting state $S1$ is defined, which uses all transitions and all conditions, on multi-condition transitions, at least once. The resulting state transitions and corresponding outputs are depicted in Table 3.5 and Table 3.7 respectively[3]. Despite the different outputs in $S2$ and $S3$, the output reaction is identical, just like the before-mentioned state transition reaction.

Both state machines calculate the next state depending on their present state and input. However, the *Mealy* machine output depends on both the present state and input while the output of the *Moore* machine only depends on the present state. This example shows when considering a certain input sequence and analyzing the output sequence, that a *Mealy* machine can be transferred into a *Moore* machine and vice versa. However, it is very important to note, that their reaction time, in terms of input to output propagation delay is different. Since the *Moore* concept only sets the output after reaching a certain state, the *Mealy* machine, in general, can react faster to a given input. The implementation of this machine within the development environment is shown in Section 3.3.

[3]In general, the specification of the initial state, in a *Mealy* machine, does not lead to an unambiguous output, since the transition defines the output. However, in this case, the transition to $S1$ always leads to an output of 0.

3.1.4.1 Mealy State Machine

George H. Mealy stated the following for his model of a sequential circuit: "A switching circuit is a circuit with a finite number of inputs, outputs, and (internal) states. Its present output combination and next state are determined uniquely by the present input combination and the present state." [Mea1955]

Table 3.4: *Mealy Transition Table (adapted from [Mea1955])*

Current State	Current Input	Output	Next State
S1	0	0	S2
	1		S4
S2	0	1	S3
	1		
S3	0	0	S1
	1		S4
S4	0	0	S2
	1		

Table 3.5: *Mealy Input-Output Mapping*

Input	1	1	0	0	0	1	1	0	-
Mealy State	S1	S4	S2	S3	S1	S2	S3	S4	S2
Mealy Output	0	0	1	0	0	1	0	0	-

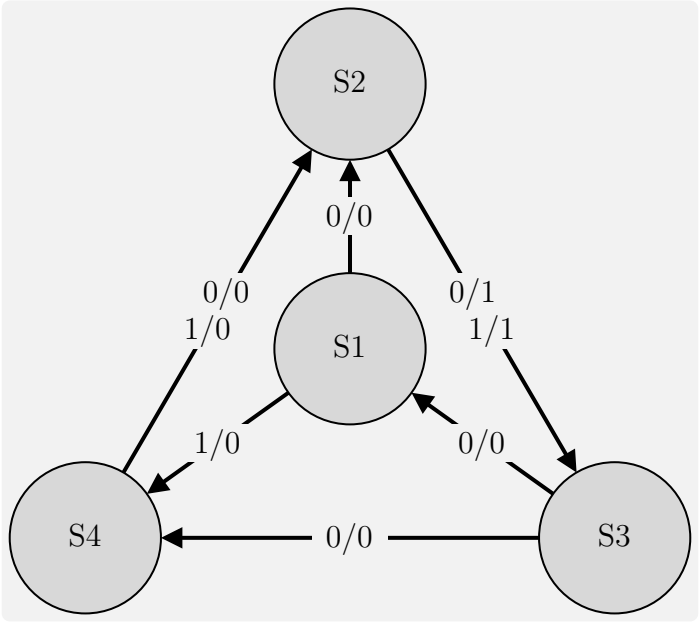


Figure 3.7: *Mealy State Machine (adapted from [Mea1955])*

3.1.4.2 Moore State Machine

Edward F. Moore stated the following for his model of a sequential machine: "The state, that the machine will be in at a given time, depends only on its state at the previous time and the previous input symbol. The output symbol at a given time depends only on the current state of the machine." [Moo1956]

Table 3.6: *Moore Transition Table (adapted from [Moo1956])*

Current State	Current Input	Next State
S1	0	S2
	1	S4
S2	0	S3
	1	
S3	0	S1
	1	S4
S4	0	S2
	1	

State	Output
S1	0
S2	1
S3	0
S4	0

Table 3.7: *Moore Input-Output Mapping*

Input	1	1	0	0	0	1	1	0	-
Moore State	S1	S4	S2	S3	S1	S2	S3	S4	S2
Moore Output	-	0	0	1	0	0	1	0	0

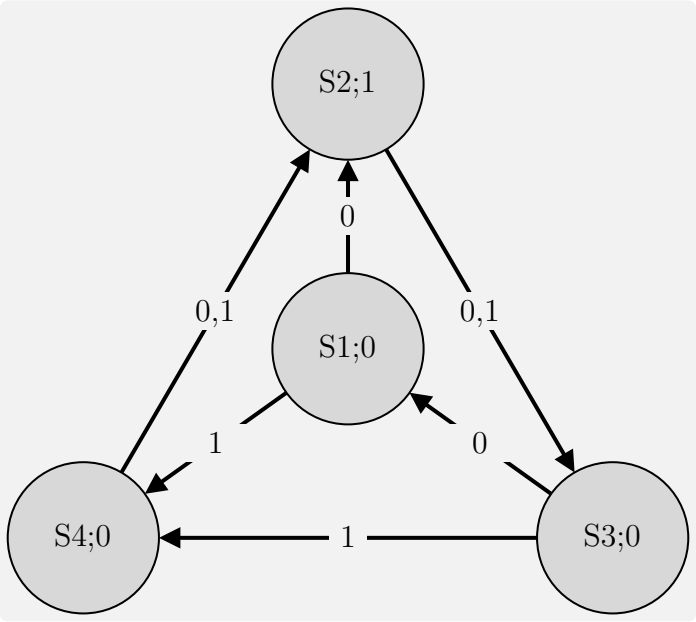


Figure 3.8: *Moore State Machine (adapted from [Moo1956])*

3.2 Design

The previous section, Section 3.1, introduced the theoretical basics that this methodology is based on. This section presents the design aspects of the methodology, which include automation challenges in general and in aviation specifically, the design steps of the software implementation, the decision logic used for the state machines, and the hierarchical decomposition structure, which is used to reduce complexity and enable complete testability.

In the beginning, Subsection 3.2.1 presents typical automation challenges in aviation. According to *Charles Billings*, complexity, brittleness, opacity, and literalism are the main challenges, higher-level modern aviation automation has to solve. Those four challenges are introduced, an aviation example is presented and the mitigation approach used in this methodology is outlined.

Following the challenges, the design steps of this methodology are presented in Subsection 3.2.2. This eight-step process is used to design the higher-level automation systems described in this thesis. It ranges from the basic definition of operating modes and grouping into different levels, over selecting necessary command injection points in the cascaded control loop depending on the control strategy, to testing and verification of the developed functions, both standalone and fully integrated.

The next two subsections, Subsection 3.2.3 and Subsection 3.2.4 cover the internal and external decision logic used to develop the automation. Multiple combined state machines, each containing their own decision logic, are used for the implementation of complex functions. However, the decision logic does not have to be part of the individual state chart itself. The state machines are used to perform mode decisions only. Therefore, the external decision logic is necessary for more complex calculations, which in turn are necessary to make certain mode decisions. Additionally, one developed submodule for checking intermittent ranges is presented and an enhanced implementation of temporal logic is shown.

The last subsection, Subsection 3.2.5, presents contribution *C1.1 - Hierarchical decomposition design strategy, minimizing complexity and optimizing testability.* It describes the generic design of this hierarchical decomposition strategy with its main parts and how it can be repeated on however many levels are necessary. This includes the conditioning of the input, the state machine itself, the routing to the next level, and the output allocation. Additionally, the injection of the generated signals into a generic cascaded control loop is presented.

Following those different aspects of the design, the next section, Section 3.3, is presenting the specific implementation. For a detailed presentation on the explicit implementation in *MATLAB* and *Simulink* refer to Subsection 3.3.3.

3.2.1 Automation Challenges

The main goals of aviation automation, in general, are increased safety and better economics [WC1980, p. 3ff]. Furthermore, automation is necessary for aviation to advance, but at the same time, the costs (in terms of undesired behavior or deficiencies) of automation need to be properly addressed as well.

According to *Charles Billings*, the main challenges, higher-level modern aviation automation has to solve are complexity, brittleness, opacity, and literalism. Additionally, training the pilot or flight crew on how to interact with the automation in certain situations is very important. Furthermore, adequate information or feedback is required for the pilot to create the possibility to make the right decisions and to understand the automation. The following list is summarized and adapted from [Bil1996, p. 89ff].

- Complexity - the user of a system is not able to understand the automation and build a mental model of the operational procedures
- Brittleness - algorithms which appear to work reliable under normal circumstances but fail completely in unusual conditions
- Opacity - lack of transparency due to insufficient information from the automation or inadequate announcement to the operator
- Literalism - automation is executing its predefined instructions without checking the possibility and applicability to the current situation

In general, the control and guidance of a remotely piloted vehicle are more difficult than normal onboard operation. Those above-mentioned problems are amplified by the physical separation of the operator or pilot in the use case of unmanned or remotely piloted vehicles. This separation leads to a reduced amount of information about the aircraft's environment. Because data needs to be collected by onboard sensors, transferred via the data link, and displayed in a Ground Control Station (GCS) to the operator, their rate and/or amount is greatly reduced. Additionally, their quality is reduced as well due to the latency of the clocked and delayed signal. This applies in particular to visual, vestibular, and auditory information, which are severely affected by the physical separation and resulting sensory isolation. [MW2004, p. 1] [Erp2000, p. 4]

This section introduces the four main challenges as listed above and described by *Charles Billings* [Bil1996, p. 89ff]. Additionally, to allow for a better understanding, a few examples, inspired by *Charles Billings* [Bil1997, p. 298ff], of those drawbacks of software functions are given in the context of aviation automation. Furthermore, the solution approach for the challenges in the developed methodology and automation functions is presented. It has to be noted, that it is not possible to completely solve those problems in general or for every application. Nevertheless, it must be ensured that their impact is as low as possible to provide robust automation functions in the context of experimental aviation automation.

3.2.1.1 Complexity

The development of higher-level automation has made aviation functions increasingly capable and more and more flexible, but at the same time, the level of complexity has greatly increased as well. Modern flight management automation has numerous modes of operation for different control axes of the aircraft. In many cases, the pilot is not able to create a mental model of the system which results in less mode awareness and can even lead to mode confusion. This is supported by successive and "hidden" mode changes, which are triggered automatically and paired with an insufficient display or indication method. Furthermore, many automation systems don't have integrated contingency procedures, which leads to less or no assistance at all when it is needed the most.

The Flight Management System (FMS) of a modern aircraft can be considered a highly complex system with countless operation modes. The reaction of pilots to such complex automation can unintentionally adversely affect the consequences. This happens if the system shows an, from the pilot's viewpoint, undesired behavior. Due to the interaction with the system and resulting behavior the pilot might revert the system to a lower level of management or completely disable it. This will happen if the pilot doesn't have a mental model and no other way of correcting the situation. Due to the deactivation of the system certain additional tasks now need to be performed by the pilot, thus increasing the workload. Furthermore, certain protection features might not be available anymore, leading to a more critical situation. [Cur1985, p. 30]

There are a lot of aviation incidents and accidents, that can at least partially be attributed to the complexity of the automatic flight control system. One of them is China Airlines Flight 140. The Airbus A300 crashed during the approach to Nagoya Airport on 26 April, 1994, only a few hundred meters away from the runway. The flight crew inadvertently activated to Take-Off/Go-Around (TO/GA) mode, in which a large amount of thrust is automatically applied and the aircraft is forced in a nose-up attitude. The pilots lowered the thrust levers and pushed down on the controls. However, they did not disconnect the autopilot, which counteracted the pilot's input with a nose-up movement of the horizontal stabilizer. This "fight" between the flight crew and the automatic system resulted in a steep climb with reduced speed, which stalled the aircraft. [Air1996]

To overcome the challenge of highly complex systems and their adverse effects, countermeasures are integrated into the design and implementation steps of this methodology. The unambiguous mode-breakdown is based on operational objectives rather than control strategies, which makes it more transparent to the pilot or operator. Grouping the modes into different levels of involvement is an additional step, that can be used if necessary. It allows for a fast reaction to critical events without identifying the exact mode.

With that approach, the developed automation functions are capable of performing the required task, while keeping negative effects from increased complexity in the design, implementation, and operation to a minimum.

3.2.1.2 Brittleness

As aviation automation becomes more and more complex, the required effort to verify that the software is working correctly under all circumstances grows excessively. This is mostly due to almost infinite combinations of input signals or variables. In this context, brittleness is an attribute of software that works well under nominal conditions but fails as soon as non-nominal events occur.

Air France Flight 296 is one example where brittleness has at least contributed to its crash. On the 26 June, 1988, it performed a low pass over the airfield at Mulhouse-Habsheim Airport as part of an airshow. The A320 descended to an altitude of about $10m$ (instead of the intended $30m$) and could not initiate a climb before hitting trees at the far end of the runway. [Bur1989]

The *High Angle of Attack* protection is used to protect the aircraft from stalling. Whenever a certain angle of attack (α_prot) is exceeded, the control mode for the elevator is switched to protection mode, in which the angle of attack is proportional to the side stick deflection until a given maximum (α_max). [Air1987b, p. 22]

Additionally, the auto thrust system normally uses the *Alpha Floor* protection, which is applying maximum thrust, TO/GA, whenever a certain angle of attack threshold (α_floor) is exceeded. This system is available from takeoff until approach when descending below $100feet$. [Air1987a, p. 143]

However, due to the very low altitude and extended landing gear, the automation disabled the *Alpha Floor* protection. This is necessary to allow the aircraft to land but would in all other cases have automatically applied full power. The *High Angle of Attack* protection prevented the aircraft from stalling during the low speed and low altitude pass over the airport, however, due to the disabled the *Alpha Floor* protection the automation did not apply full thrust, which would have probably prevented the crash.

The methodology developed in this thesis is used to design software that is used in an experimental environment. In one case it is even used on the maiden flight of an experimental aircraft. In this context, the possibility of non-nominal events is even higher than in other aviation applications. Therefore, the design and implementation must be very robust and avoid brittleness. This is accomplished by using decision logic based on deterministic finite state machines with robust transition conditions that are thoroughly tested, using formal methods, to ensure correct behavior even in unexpected environmental conditions.

Additionally, the above-mentioned level structure supports higher-level automation functions with reduced complexity, which in turn leads to a clearer design with a lower error possibility. Furthermore, extensive testing and formal verification methods are used to ensure a valid design, correct implementation, and guaranteed properties of the automation under arbitrary operation conditions.

3.2.1.3 Opacity

Another challenge of higher-level automation, especially in aviation, is transparent behavior. In a survey of pilot attitudes toward automation, their frequent responses to automation systems can be paraphrased with the following questions [Wie1989].

- What is it doing?
- Why is it doing that?
- What is it going to do next?

The software needs to provide answers to these questions and they need to be shown in an adequate and easily decipherable way. If one or both of these tasks are not fulfilled, this lack of transparency is also referred to as opacity.

Besides problems directly caused by the pilot, opacity can be explained by two main contributing factors on the system or software side. One is insufficient or too complex information from the automation. Then the pilot is not able to build a mental model of the automation and cannot comprehend the automation actions and motivations. Another problem is an opaque interface between the pilots and the automation, which makes it difficult to track its state and activity. [SW1994]

Air Inter Flight 148 crashed on 20 January, 1992, after the pilots missed a small but very decisive point, in the literal sense of the word. On approach to Strasbourg Airport in France, the flight crew unintentionally commanded the autopilot to descent at a vertical-speed of $-3300 feet/min$ instead of the correct $-3.3°$ flight-path-angle. Although this being a pilot error, opacity has contributed to the fact that the mistake was not discovered in time. The display was showing -33 (indicating $-3300 feet/min$) instead of the desired -3.3 (indicated as $-3.3°$). [Fra1993]

A single dot was used to distinguish the sink rate from the flight-path-angle. A sink rate of $3300 feet/min$ at the approach speed of an Airbus A320 corresponds to about $-14.5°$ flight-path-angle and had catastrophic consequences.

The breakdown to different modes based on operational objectives and grouping into different levels of involvement helps to prevent the opaqueness of the automation. Additionally, the software provides sufficient information on not only the current operation mode and level of involvement but also respective parent modes or superimposed mode functions. Furthermore, all platforms use custom displays for the pilot and/or custom interfaces for the operator in the GCS.

With this set of information, the pilot or operator knows what the system is doing and can understand why it is doing that. The question of what is it going to do next can be answered with comparatively little training, which is supported by the mode and level structure as well as computer simulations.

3.2.1.4 Literalism

In contrast to human problem solving, the automation algorithm is constrained by its instructions. A Human can draw knowledge from any relevant source to support the reaction in unusual circumstances. The automation, on the other hand, will only react according to the programmed functions. In general, it cannot "learn" or find different solutions and is therefore insensitive to unanticipated events.

This "narrow-mindedness" of software, in general, is also referred to as literalism. It especially occurs when automation functions are designed to perform a certain task but do not analyze if the goal of that task is achievable. Therefore, the workload of the human operator will be low during normal operation, but greatly increase when unexpected events, with respect to the designer of the software, occur.

On June 6, 1994, an A320 operated by Hong Kong Dragon Airlines was on an approach to runway 13 at Hong Kong International Airport, Kai Tak. The aircraft was at $800 feet$ when it encountered a severe gust, which caused the trailing edge flaps to be locked in the fully down position (40°) and triggered the pilots to go around.

The second and third approaches were performed with the flaps-lever being in positions for 20° and 10° respectively. However, both were abandoned due to lateral oscillations and roll-angles of up to ±30°. During the 4th approach, with the flaps-lever at the position for 20°, the same oscillations occurred, however, due to the critical fuel state of the aircraft, the pilots continued towards the runway and touched down. The aircraft came to a stop on a parallel taxiway.

The investigation revealed, that the critical behavior of the aircraft was caused by a mismatch of the sensitivity of the lateral control law and the actual flap position. The sensitivity was taken from the position of the flap-lever, rather than the real position of the flaps because a deviation was presumably not considered during implementation. Besides other factors, like missing crew coordination and delay of effective directional control after touchdown, the literalism of the software contributed to this incident. [Acc1997]

In the context of higher-level system automation using finite state machines, this can occur if a mode has a certain requirement to be entered and continued. If however it is implemented in a way that the condition is only examined on entry and a falsification of the condition does not lead to a mode change the automation is creating literalism.

Another problem can occur if concatenated modes with intermediate transfer conditions are used. If the transition does not occur in the anticipated way the entry condition of the intermediate mode can be false, while the exit condition is true. This leads to a stuck automation because one, possibly unimportant, transition did not take place as the designer had expected.

The challenge of literalism is addressed in the system automation by self-monitoring capabilities, automatic contingency maneuvers, and a design and implementation methodology, which requires a comprehensive evaluation of entry and exit conditions as well as redirections for concatenated modes.

3.2.2 Design Steps

The design of higher-level system automation developed within this methodology is based on an eight-step process. Depending on the developed function and complexity of the system it is not always necessary to complete all of them. Each step is described in the following paragraphs.

Step 1: The first step is the definition of nominal as well as non-nominal operation modes. The nominal modes are defined based on their operational objective (e.g. "parking" or "waiting-for-trim-point"). Non-nominal modes can be divided into two groups. Contingency modes, that are based on mission changes (e.g. "return-to-base" or "loiter") and contingency modes due to malfunctions (e.g. "link loss" or "Global Positioning System (GPS) loss").

Step 2: The next step is only necessary if the number of different modes is too high or they are too dissimilar in another way. In this case, they are grouped into appropriate parent modes and consequently placed on different levels. Therefore, one mode can have one parent mode and multiple child modes. This is repeated until the number of modes on each level is below a suitable threshold, usually around five. Modes can be split based on the overall status of the system (e.g. "active" or "standby"), based on the user (e.g. "flight operator" or "external pilot"), based on the existence of a malfunction (e.g. "normal" or "link loss"), and many more. Additionally, consecutive modes that are automatically sequentially activated are consolidated within one parent mode and placed on the respective lower level. Separate mode calculations (e.g. loiter) are implemented in parallel state machines, so they do not interfere with the main one.

Step 3: Then an appropriate control strategy is specified for each non-parent mode. Parent modes don't need a control strategy, because the control strategy is specified in the child modes (as long as they don't have child modes of their own at which point they are parent modes as well). The control strategy consists of a selection of control variables for each axis or combination of axes (e.g. vertical: altitude and lateral: heading). Furthermore, the corresponding control module needs to be selected. Depending on the mode it can be necessary to use different cascaded control loops at the same time (e.g. superimposing altitude commands from the autopilot while continuing laterally based on the waypoint module).

Step 4: Depending on the necessary control modules and commands, from the previous step, injection points to the different control modules are defined. Within the cascaded control loop, the commands for the respective next (lower) loop are created by the adjacent earlier (higher) loop. The injection points are placed between the modules and act as a signal routing layer or switch. Depending on the required function they need to be able to switch or modify some or all commands, which are passed to the next module. The capabilities range from forwarding and/or modifying the original signals to replacing them with self-generated or externally provided commands.

Step 5: For each mode with a control strategy the respective commands or source of commands must be defined or selected. The commands can be externally specified (e.g. commands from a user), generated by another control module or sensor (e.g. flight path deviations for the trajectory module), or internally generated (e.g. a generated flight test maneuver for the actuators) by the automation. If the commands are externally generated and don't need to be monitored they can be directly connected to the injection point, otherwise, the monitoring of the command is done within the specific mode and the monitored command is then forwarded to the respective injection point. The same principle applies if the command is generated by one of the other control loops. If the commands are generated within the automation itself, they need to be parameterizable without changing the internal structure, to support different platforms.

Step 6: In preparation for the next step all possible mode changes need to be identified. Due to the small number of modes on each level, it is relatively easy to analyze all possibilities. Following the identification of all mode transitions, the conditions for each transition need to be formalized. The decision logic used for the transition condition in the finite state machine is based on boolean algebra. Depending on the complexity, the condition can either be calculated in the state machine or externally. An external calculation can be necessary if the conditions consist of numerous logic gates, temporal logic, or complex calculations requiring a separate state machine or subsystem. Detailed information about the decision logic used in those transitions is described in the following subsections (Subsection 3.2.3 and Subsection 3.2.4).

Step 7: Each of the previously defined transition conditions requires a respective transition action. They need to be derived to execute internal (within the automation) as well as external (with respect to the control loops) actions. Internal actions can consist of propagating mode changes to other levels, triggering calculations in certain steps, or performing simple calculations within the state chart. Transition actions needed for external interaction might consist of defining switch positions of the injection points, commanding specific values, and activating or deactivating features of other control modules.

Step 8: The last, but very important step is the testing and verification of the developed higher-level automation function. This testing process is divided into different levels of testing. Each of which is targeted to a different test environment or system under test. The tests include Unit Tests and Model Checking. Additionally, Model in the Loop (MiL), Software in the Loop (SiL), Hardware in the Loop (HiL), and Aircraft in the Loop (AiL) simulations are used. Ground Tests and Flight Tests prove the real-life applicability of the software. Depending on the test, they are either standalone tests for the developed automation functions or integrated software tests using the other control modules and a Flight Dynamic Model (FDM). Some of the standalone tests use formal methods to create deterministic state machines with guaranteed properties under arbitrary conditions. Detailed information about the testing process, the different levels of testing, and the test environments are given in Section 3.4.

The eight previously described design steps of the proposed methodology for the higher-level system automation are summarized in the following.

Step 1 Definition of nominal modes based on their operational objective and contingency modes based on the aircraft's reaction or possible malfunctions

Step 2 If necessary, consolidation of modes into different levels based on the status of the system, type of operator, malfunction detection, etc.

Step 3 Specification of an appropriate control strategy and selection of the respective control module or combination of control modules

Step 4 Definition of necessary command injection points on the different layers of the cascaded control modules of the Flight Control Computer (FCC)

Step 5 Selection of required commands for each mode and their respective injection point and alteration method within the cascaded control loop

Step 6 Identification of all possible mode changes on all levels of the automation and formalization of the respective transition condition

Step 7 Derivation of necessary transition actions, both within the automation as well as with respect to the commands for the external control loops

Step 8 Testing and verification of the developed automation functions as standalone systems as well as fully integrated with the other modules

With this approach, an automation can be designed that utilized various functions of a cascaded control loop to perform higher-level system procedures. The design steps create distinguishable modes on different levels. Further steps reduce the complexity of the system during the design and the implementation as well as during the testing and using while supporting sophisticated designs with a high number of operating modes. A thorough testing process ensures robust functionality under all conditions.

3.2.3 Internal Decision Logic

For the applications presented in this thesis, all decision logic is implemented in state machines and based on Boolean algebra as introduced in Subsubsection 3.1.2.1. The state machines are used for mode decisions only and not for mathematical computation of commands and the such like. The latter is performed in conditionally executed subsystems, which are introduced are Subsection 3.2.5.

However, in some cases, even the logic needed to perform mode decisions is not directly implemented within the state machine. Depending on the complexity, the logic is implemented within the state machine and therefore using *Stateflow* or outside of the actual state machine in *Simulink*. This subsection is focusing on the internal decision logic in *Stateflow*, while the next is focusing on external decision logic in *Simulink*. For a better comprehension of *Simulink* models in the following sections, all block names are shown. Default block names would normally be hidden during implementation.

Finite state machines are triggered by discrete events, activating a transition that leads from one state to another. Those events are derived from decision logic that depends on continuous or discrete signals. However, in all practical applications that are based on a clocked processor, the decision logic depends on discrete signals. This needs for example be considered when comparing signals to thresholds.

Relational operators are used for comparing two signals or more precisely two values (of different signals) at a specific point in time. This can either be a comparison of one signal to another or to a predefined fixed value, also referred to as threshold. In both cases, their output is a Boolean expression being either `true` or `false`. Relational operators are listed in the top part of Table 3.8.

Additionally, logical operators can be used to create Combinational Circuits (c.f. Subsubsection 3.1.2.1). Even though those operators concatenate two or more relational comparisons, their result is also a Boolean expression. The respective operators, available in *Stateflow*, are listed in the lower part of Table 3.8.

Table 3.8: *Relational and Logical Operators*

Type	**Operator**	**Meaning**
Relational	==	Equal to
	! =	Not equal to
	>	Greater than
	<	Less than
	>=	Greater than or equal to
	<=	Less than or equal to
Logical	&&	And
	||	Or
	!	Not

3.2.4 External Decision Logic

In most cases, the decision logic is designed with various relational and logical operators in *Stateflow*. However, in some cases, the external calculation in *Simulink* is easier, cleaner, and reduces complexity inside the state machine itself. One example might be the comparison of a signal to a threshold, which is needed in multiple transitions, that can be replaced by a *Simulink* implementation, which only forwards the resulting Boolean expression to the state machine.

In some cases, the relational or logical comparison is also implemented in *Simulink* to keep the general validity of the state machine. This can be the case if the same state machine is used multiple times on signals with different names.

In this subsection, the used *Simulink* Blocks are introduced and encapsulated implementations for custom functions are presented. An Overview is listed in the following.

- Simulink Blocks
  - Relational Operator, Compare to Constant
  - Logical Operator
  - Detect Increase, Detect Decrease, Detect Change
- Custom Implementation
  - Intermittent Range Check
  - Temporal Logic

3.2.4.1 Simulink Blocks

The *Simulink* equivalent blocks to the previously introduced operators (c.f. Subsection 3.2.3) are depicted in the top row of Figure 3.9.

In the lower row, specific blocks to detect an increase, decrease, or any change in a signal are shown. While this condition can also be implemented in *Stateflow*, its respective implementation in *Simulink* is much easier and cleaner. Thus only the result of this evaluation is forwarded to the state machine.

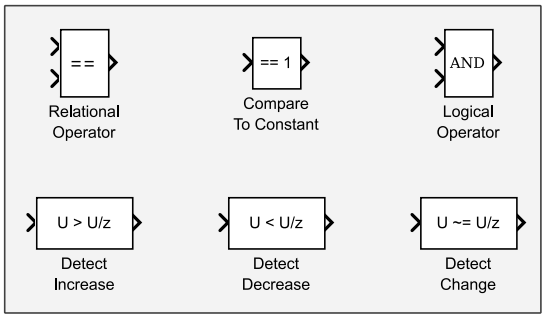


Figure 3.9: *Simulink Decision Logic Blocks*

3.2.4.2 Intermittent Range Check

The operational modes used in the applications described in this thesis, make use of representing modes as enumerated values. Therefore, it is necessary to check for valid modes in various instances. This results in the necessity to check an enumerated mode for its valid range, which is not necessarily continuous. Therefore, the Intermittent Range Check is developed, in which the enumeration is checked against a valid range.

In the following, an example is used, which uses integers instead of enumerations for easier understanding. In the application, where enumerations are used they need to be converted to integers beforehand for *Simulink* to be able to handle them.

In the following example, the input has been sequentially increased from one to six, which results in the situation shown in Figure 3.10. Within the developed block, shown in Figure 3.11, the vector, which includes the valid range is compared to the current input. If the input is valid the comparison leads to a vector with one `false` element. The following logic is used to find this one element, which results in the switch being in the `false` position. If the input is outside the valid range, all elements of the comparison are `true`, the switch is in the `true` position, and outputs the last valid signal.

Therefore, the Intermittent Range Check will always output a mode command, which is in the valid range, and an additional signal, which can be used to determine the validity. Both can be used within the state machine, without directly implementing such logic.

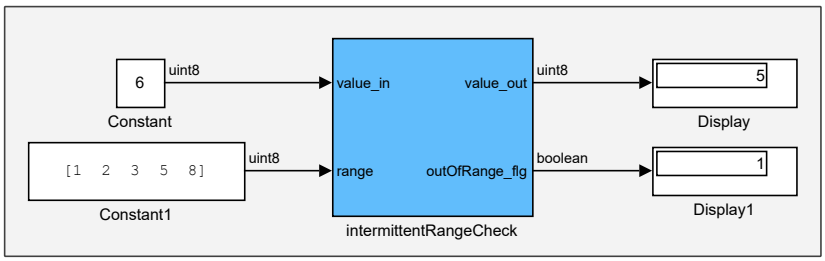


Figure 3.10: *Intermittent Range Check Top*

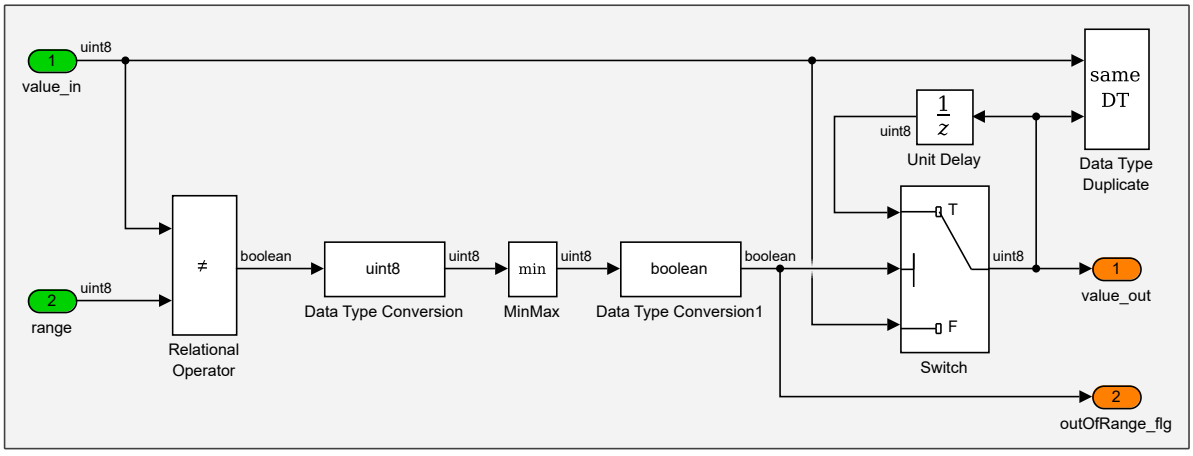


Figure 3.11: *Intermittent Range Check*

3.2.4.3 Temporal Logic

The previous relational and logical operators, as well as their combination to Combinational Circuits, can only evaluate the inputs at a certain point in time and trigger transitions accordingly. Additionally, triggering a transition based on past time might be necessary to accomplish time-based sequential states. Such an implementation is referred to as temporal logic. [Pnu1977]

Stateflow supports the direct implementation of such temporal logic in state charts. Available operators for event-based temporal logic are: *after*, *before*, *at*, *every*, and *temporalcount* with keywords: *tick*, *sec*, *msec*, or *usec* [TM2016d]. Those can be used to implement various time-based sequences and transitions. However, due to guidelines for model-based code generation, they are not used.

Instead, a discrete implementation of a counter in *Simulink* is used. In every time step, the counter is increased by one, and the variable is fed into the state chart. The counter saturates at the maximum value of the used variable type. Which in this case is $4,294,967,295$ $(2^{32}-1)$ for a `uint32` variable. For systems running at $100Hz$, this is equivalent to more than one year with a precision of $10ms$, which is sufficient for modeling sequential state for the applications described in this thesis. A reset can be issued from the state chart using the Boolean variable `reset_cfg`. The postfix (`_cfg`) of the variable, indicates an action on both rising and falling edge (c.f. Subsubsection 3.3.4.7).

In general, a timer can be used to trigger something after a certain time or to measure the time difference between two events. Due to the saturating nature of this implementation, it should only be used for the first example. A usage to measure time difference could, however unlikely considering the maximum time, result in a, potentially catastrophic, failure of the system.

With this simple replacement, all necessary temporal conditions can be implemented in a clear and simple way. An example chart using this counter is depicted in Figure 3.12. The counter is connected to the state machine with three inputs and two outputs. The `saturated_flg` can be omitted if not necessary.

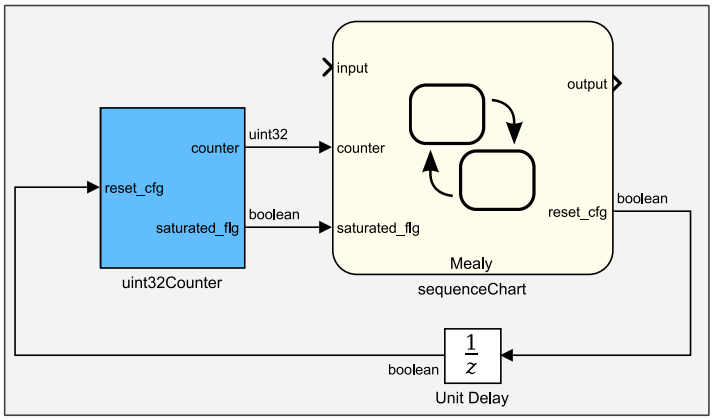


Figure 3.12: *State Machine with Temporal Logic*

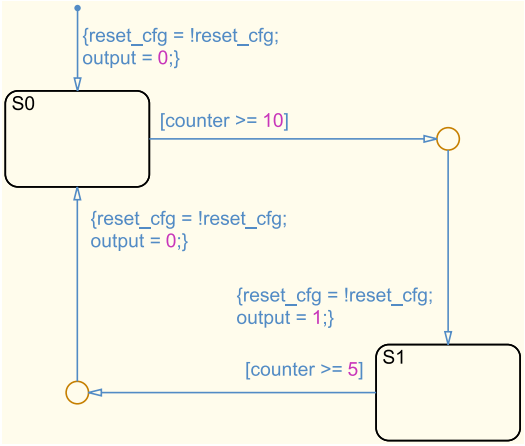


Figure 3.13: *Sequence Chart*

An example of the usage of such a counter in *Stateflow* is depicted in Figure 3.13. Furthermore, the content of the counter itself is shown in Figure 3.14.

As shown in the state chart the counter can be reset with a single line. Using `reset_cfg = !reset_cfg`, which sets the boolean output `reset_cfg` to its respective opposite value, resets the counter to zero. After such a transition action the counter can be used to delay the transition to another state.

In the *Simulink* implementation the switch, which resets the counter, and the counter itself becomes apparent. Additionally, the saturation, which prevents an overflow of the variable, and the generation of the respective flag is shown. In this implementation, the saturation is set to $4,294,967,294$ $(2^{32} - 2)$ to result in the maximum range after the addition.

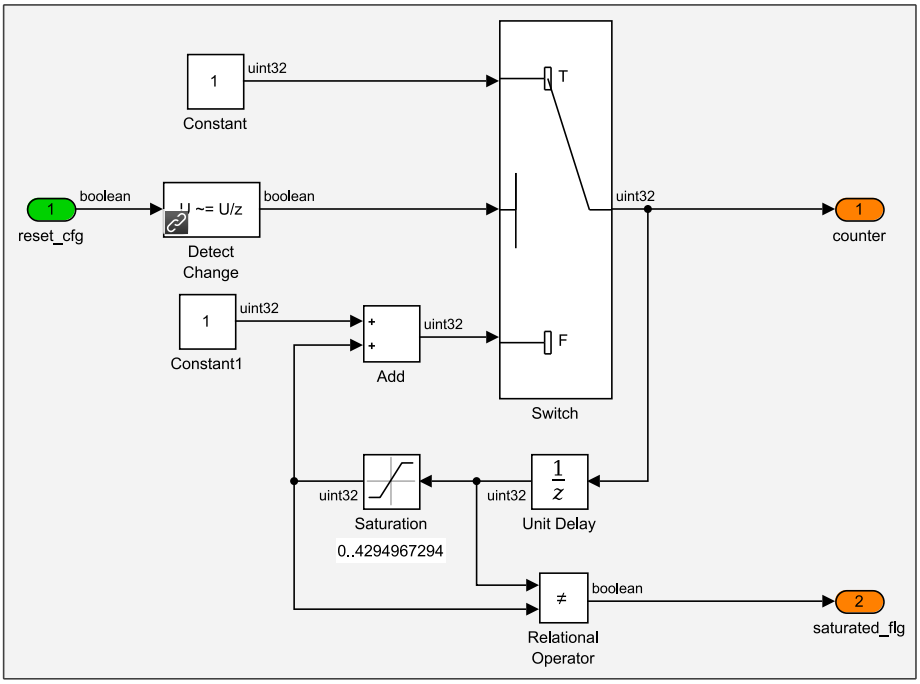


Figure 3.14: *Discrete Counter in Simulink*

3.2.5 Hierarchical Decomposition Structure

This subsection is introducing contribution *C1.1 - Hierarchical decomposition design strategy, minimizing complexity and optimizing testability.* Additionally to the generic design in this subsection, Subsection 3.3.3 describes the specific implementation in *Simulink* and *Stateflow*. An Overview of the schematic is depicted in Figure 3.15.

The main goal of this design and implementation strategy is to minimize the complexity of higher-level automation state machines and optimize their testability. It is based on decomposing the overall functionality into a hierarchical structure with modular atomic units on different levels. This reduces the complexity on each level and enables separate development, implementation, and testing.

Besides the definition of the structure itself, this strategy includes the definition of two signal buses, which include all necessary signals in all levels of the automation. On the one hand, the `input_bus` includes all necessary external inputs. They may consist of commands from the Flight Operator (FO), measurements from sensors, or calculations from other flight control modules. On the other hand, the `output_bus` consists of calculated values that are used either within the automation itself or externally. The internal values are typically the states of all state machines, while the external values may consist of commands for other modules, information for the GCS, or direct commands for the control surfaces.

Each level consists of four parts: the *Input Conditioning*, the *State Machine*, the *Routing*, and the *Output Allocation*. This is depicted in Figure 3.15. The schematic is reduced to the essential parts of the hierarchical decomposition structure. Additional computations or secondary state machines are omitted for the sake of clarity.

The *Input Conditioning* is the extraction of necessary signals from the `input_bus` and possible range and validity checks. This is done to create the explicit visibility of relevant variables on each level. *Stateflow* is capable of using buses as input and output. However, the developed guidelines prohibit the use of buses as direct in- or output to charts to guarantee the visibility of the relevant variables only.

The *State Machine* is calculating the operating mode with respect to the current level. On the highest level, those may consist of *Operational* or *Standby*. Additional calculations and even other state machines can be used in parallel to this main state machine, but are omitted in the schematic.

Information about the current operating mode from the state machine is then forwarded to the *Routing* part of this first level. Within this *Routing*, the respective subsystem, depending on the previously calculated operating mode, is executed. This is done using conditionally executed structures like "if-else" or "switch-case", which only executes the necessary parts of the automation.

In the end, the *Output Allocation* is used to assign calculated values to the `output_bus`. This includes values from this level and all lower levels.

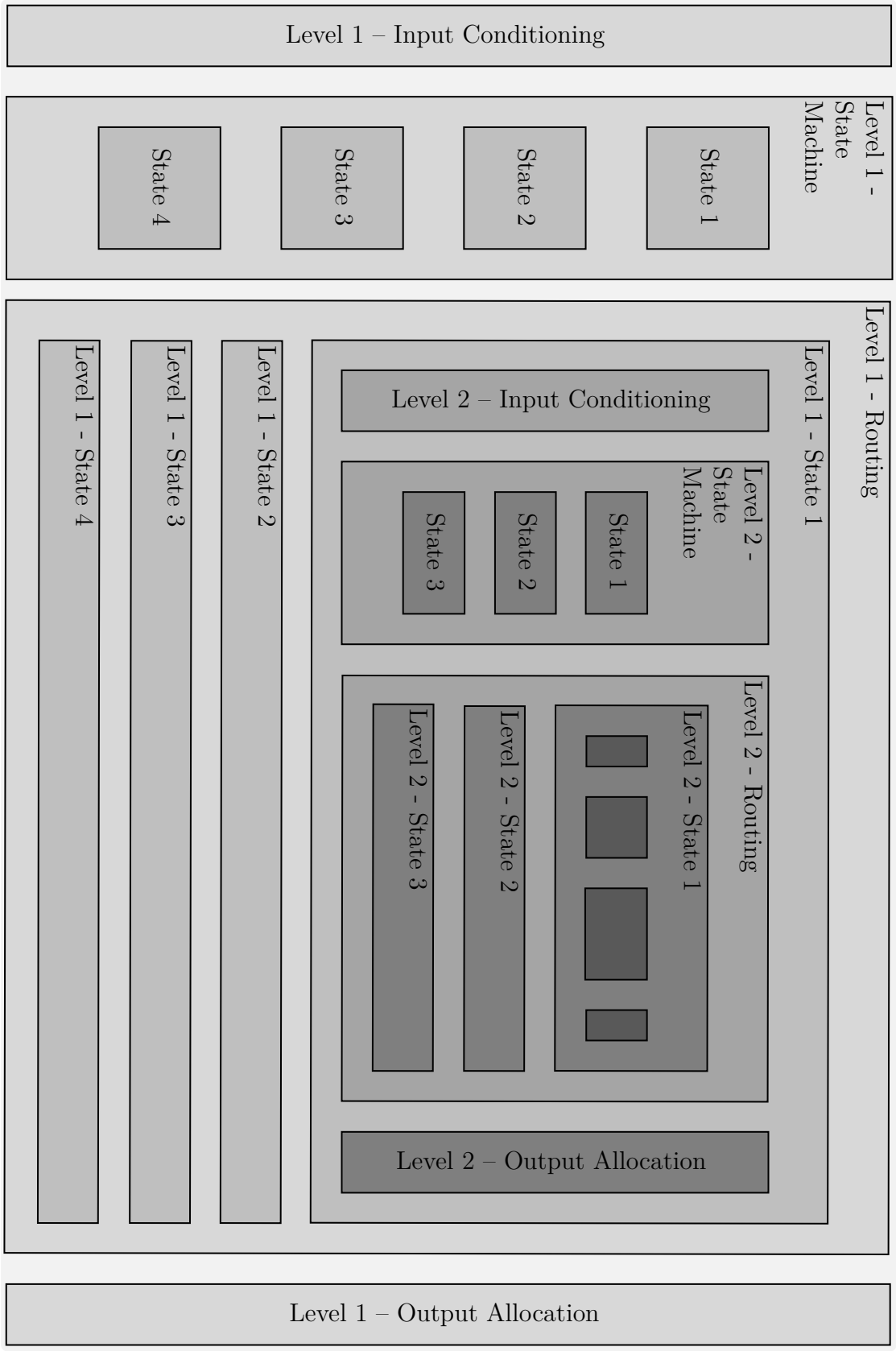


Figure 3.15: *Hierarchical Decomposition Scheme*

The hierarchical decomposition structure with three levels is shown in Figure 3.15. Depending on the complexity of the overall automation, the decomposition can be expanded to as many levels as necessary. Additionally, on each level, the *Routing*, parallel computations, or secondary state machines can be grouped into subsystems to generate encapsulated functions, which can also be tested separately.

Besides this strategy to reduce the individual complexity of the automation, it is necessary to interact with the rest of the flight control system. The injection architecture which is used is depicted in Figure 3.16. This simplified example shows a generic control loop with three cascaded control loops and the administrating automation.

Each control loop has its own control switch subsystem, which is actually part of the automation. Those systems consist of multiple inputs (marked as lines on the left) and can also include additional calculations. The automation that is controlling those switches can, therefore, control the input to every control loop without passing all signals through the state calculating part of the automation. Each switch can have multiple inputs, which can be used as commands for the respective control loop. Those can either be commands from the next higher adjacent control loop, commands generated within the automation, or commands from the adjacent control loops that have been modified by the automation. In the latter case, this alteration takes place in the switch subsystem itself. Examples of this injection architecture are shown in Figure 4.5 and in Figure 5.4.

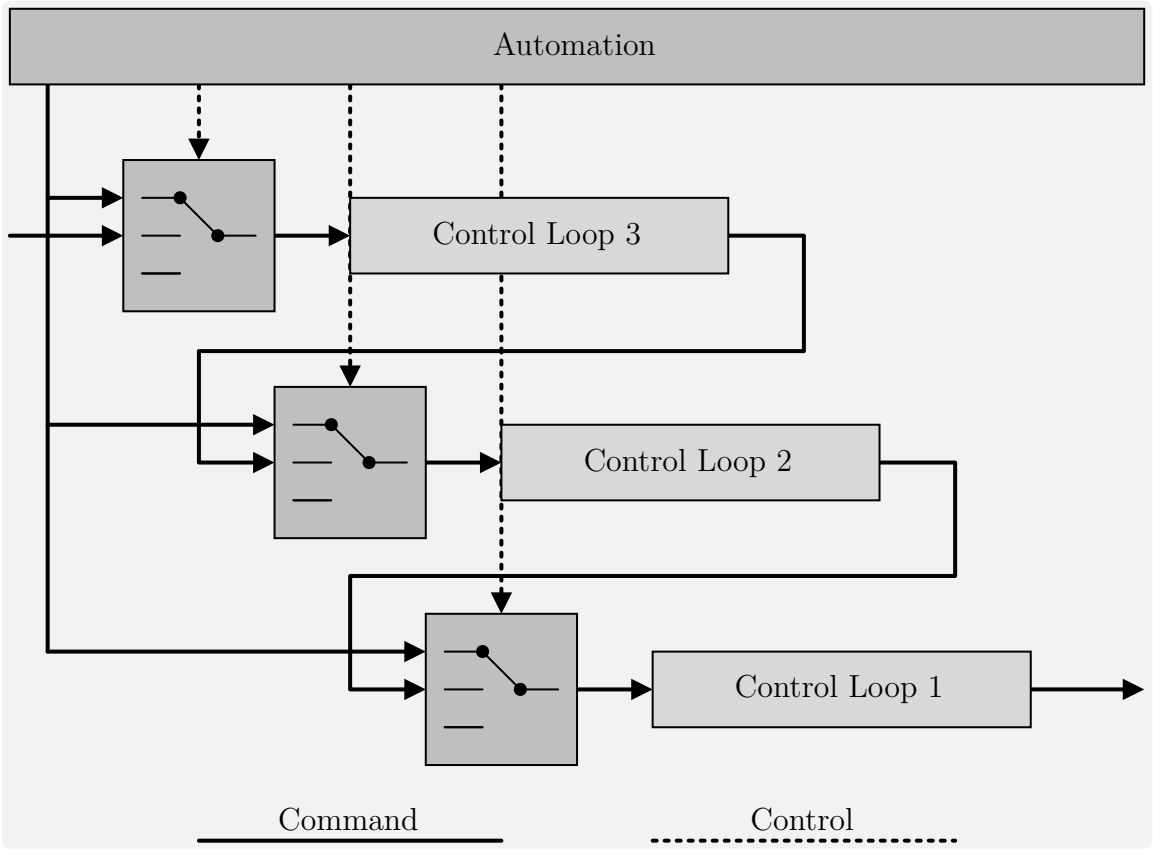


Figure 3.16: *Injection Architecture*

3.3 Implementation

Following the last section, Section 3.2, which presented the different aspects of the design process of this methodology, this section provides a detailed description of the implementation. This includes the toolchain used for the design, implementation, and code generation, and the necessary *Stateflow* elements to create the state machines. Additionally, the implementation of the hierarchical decomposition structure and developed guidelines, which improve various aspects are presented.

The first subsection, Subsection 3.3.1, introduces the toolchain, which is used for the design, implementation, and code generation of the developed functions. This includes *MATLAB*, *Simulink*, *Stateflow* as well as the code generation process.

In the following, Subsection 3.3.2 presents the *Stateflow* elements, which are used to create the state machines. In general, various elements and settings can be used in *Stateflow*. However, many of them have a negative impact on readability, workflow, verification, or code generation and are therefore prohibited by the developed methodology.

The generic hierarchical decomposition strategy is introduced in Subsection 3.2.5. In this section, Subsection 3.3.3 presents the explicit implementation of this strategy using *MATLAB*, *Simulink*, and *Stateflow*. This includes an example highlighting the major parts of this level structure as well as the presentation of the command injection implementation into the control loops.

The last subsection of this section, Subsection 3.3.4, is presenting the contribution *C1.2 - Modeling guidelines for implementation, minimizing opacity and maximizing software maintainability*. It provides an overview of all developed, state machine-related guidelines that are part of this methodology and used for the design and implementation of the automation functions presented in this thesis. Each guideline is presented with a specific category and rationale, explaining its purpose.

Following the implementation aspects in this section, the next section, Section 3.4, is presenting the testing and verification process of this methodology.

3.3.1 Toolchain

This subsection introduces *Stateflow* and the rest of the toolchain used in the model development process at the Institute of Flight System Dynamics (FSD) [HSN+2017]. This includes *MathWorks*, *MATLAB*, *Simulink*, *Stateflow*, and the Embedded Coder.

3.3.1.1 The MathWorks

The MathWorks (often abbreviated to *MathWorks*) is a company that specializes in software for mathematical calculations. By their own statement, they are the leading developer of mathematical computing software for engineers and scientists. Their two major products, *MATLAB* and *Simulink* are targeted at accelerating the pace of engineering and science. [TM2016a]

3.3.1.2 MATLAB

MATLAB, one of the major products of *MathWorks*, is used to design and analyze various systems. It is, among others, used in automotive, aeronautic, and astronautic applications for signal processing, control design, robotics, and many more. [TM2016j, p. 1-2ff]

Within the model development process at FSD it is used as the development environment for the design, implementation, and testing of flight control software. This includes state machines for automation, as presented in this thesis, as well as flight controller design, simulations using FDMs, and flight test data analysis.

MATLAB is subject to an update cycle of two per year, resulting in an "a" and "b" version for each year [TM2016k, TM2016l]. The methodology and application presented in this thesis were developed for projects with real-life applicability. Consequently, the *MATLAB* version used for design, implementation, and testing was fixed during the early phases of the design and development process. This was done to avoid bugs through software updates and increase confidence in the final software used for flight tests. Therefore, the methodology and application described in this thesis are based on *MATLAB* 2016b, even though at the point of writing, newer versions exist.

3.3.1.3 Simulink

Simulink is a graphical development environment for model-based design based on *MATLAB*. It provides a graphical editor, solvers for modeling and simulating dynamic systems, and even customization block libraries. [TM2016v, p. 1-2ff]

At FSD it is used for the design and implementation of flight control algorithms. The different cascaded control loop modules are implemented separately and then combined and connected in the aircraft-specific integration model. In this setup, *Simulink* is also used for testing and the simulation of FDMs.

3.3.1.4 Stateflow

Stateflow is a development environment for modeling and simulating combinational and sequential circuits, that are introduced in Subsubsection 3.1.2.1 and Subsubsection 3.1.2.2. It utilizes state machines and flow charts to model and simulate this decision logic. The designed logic can be used for supervisory control, task scheduling, or fault management applications. The graphical representation of state transition diagrams and flow charts is used for animation during simulation and testing. [TM2016x, p. 1-2ff]

Everyday example applications in which *Stateflow* can be used include Automated Teller Machines (ATMs), Traffic Lights, or Vending Machines.

At FSD it is used for flight control modules that need to handle various states such as the *Automatic Takeoff and Landing (ATOL)*, the *Trajectory Generation (TG)*, or the *System Automation (SA)* module.

3.3.1.5 Embedded Coder

Following the design and implementation using *MATLAB*, *Simulink*, and *Stateflow*, the Embedded Coder is used to automatically generate *ANSI C-Code*. In contrast to the *Simulink* Coder [TM2016w, p. 1-2], the Embedded Coder [TM2016g, p. 1-2] includes more settings that can be used to customize the code generation. [TM2016h]

Excerpts from the automatically generated main source and header files for the example Edge Detector are shown in Listing 3.2 and Listing 3.3. They include the three main functions: `initialize`, `step`, and `terminate`. The function `initialize` is used in the very first execution step to initialize necessary variables. Following this first execution, the function `step` is executed in every subsequent step. The third function `terminate` can contain closing actions for the last execution. However, within the flight control functions, presented in this thesis, this function is never used.

After automatically generating source code from the design models, this code is merged with a manually written code framework, which is handling the inputs and outputs of the FCC. This includes the import from the physical interfaces, transferring the information to specified *Simulink* input structures, and executing the flight control algorithm. After the execution, *Simulink* output structures are used to construct physical interface messages, which are then transmitted.

```
#include "edgeDetector_types.h"
...
/* Model entry point functions */
extern void edgeDetector_initialize(void);
extern void edgeDetector_step(void);
extern void edgeDetector_terminate(void);
...
```

Listing 3.2: *EdgeDetector - EdgeDetector.h*

```
#include "edgeDetector.h"
#include "edgeDetector_private.h"
...
/* Model initialize function */
void edgeDetector_initialize(void)
...
/* Model step function */
void edgeDetector_step(void)
...
/* Model terminate function */
void edgeDetector_terminate(void)
...
```

Listing 3.3: *Edge Detector - EdgeDetector.c*

3.3.2 Stateflow Environment and Chart Elements

This subsection introduces the elements of *Stateflow* that are used to design state machines for the automation functions described in this thesis. After an overview of the *Stateflow* environment in *Simulink*, individual chart elements are presented before the previously introduced Edge Detector (c.f. Subsection 3.1.3) is used as an example. [TM2016c]

Stateflow has multiple environments that can be utilized to implement state machines in various forms. However, due to guidelines, presented in Subsection 3.3.4, only a limited set is used which ensures that various requirements are met.

3.3.2.1 Chart

The *Stateflow Chart* is one of the environments for finite state machines in *Simulink* and the one used for the methodology and applications presented in this thesis. Figure 3.17 depicts a simple integration of such a *Chart* with one input and output in *Simulink*.

Other possible *Stateflow* elements are shown in Figure 3.18. The Truth Table and Transition Table are other environments to implement state machines, while the Message Viewer (also called Sequence Viewer in newer releases) can be used to visualize messages and events between *Stateflow* and *Simulink*. However, those are not used in this thesis and are listed here only for the sake of completeness. In the following only, elements within the *Stateflow Chart* are discussed.

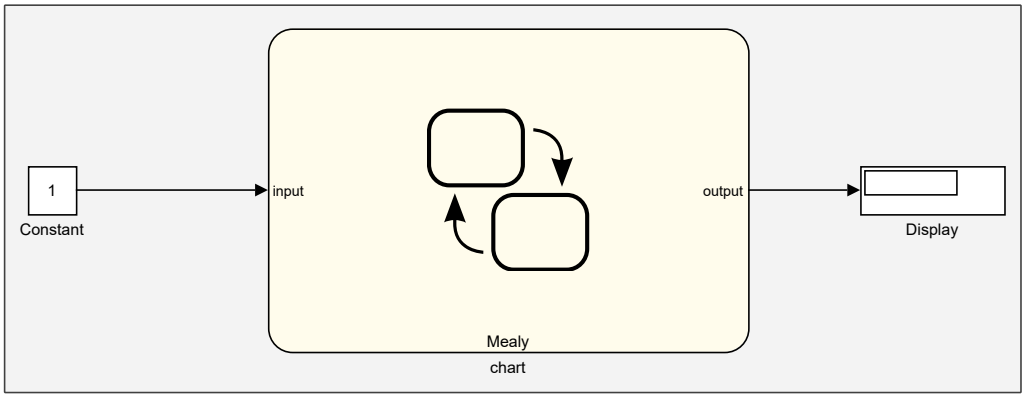
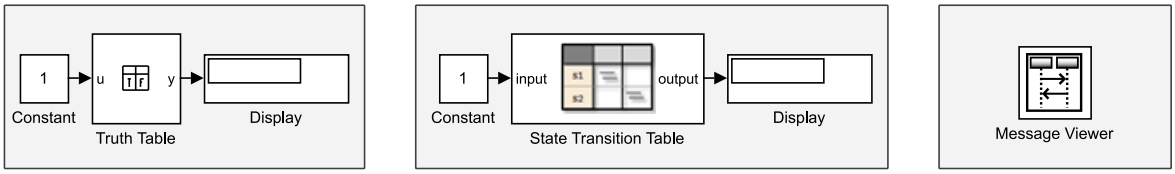


Figure 3.17: *Stateflow Chart in Simulink Environment*



(a) Truth Table (b) Transition Table (c) Message Viewer

Figure 3.18: *Other Stateflow Elements in Simulink*

3.3.2.2 State

As the names *Stateflow* and state machines imply, the *State* is the most important element in them. In *Stateflow*, those states are visualized using rounded boxes with their individual name in the top left corner.

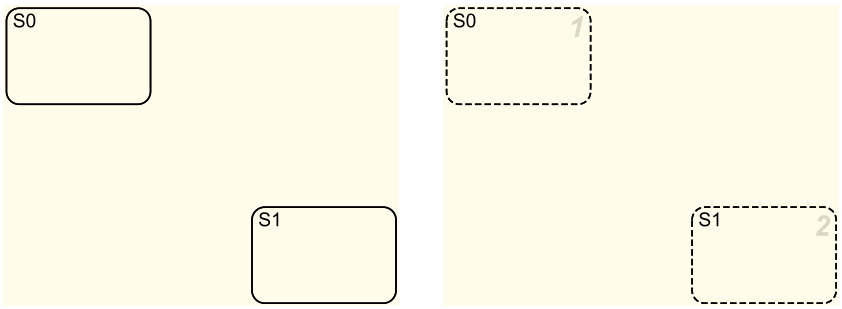
The name of the states must be unique within the *Chart*. In this case, the states `S0` and `S1` are arranged in a diagonal pattern from the top left to the bottom right. This arrangement is used to provide better connectivity between all states, which is especially useful with three or more states in one *Chart*.

The decomposition of states in *Stateflow* can either be exclusive or parallel. As the name implies, exclusive states can only be active one at a time, while parallel states can be active simultaneously. Exclusive states (shown in Subfigure 3.19(a)) are indicated by solid borders, while parallel states (shown in Subfigure 3.19(b)) are indicated by dashed lines forming their border. Parallel states have a large tendency of complicating state machines, without proving necessary features, that cannot be replaced. The developed guidelines therefore only allow the usage of exclusive states.

The two states obviously need some kind of connection. In fact, without suppressing errors for this example, *Stateflow* would immediately highlight this with a "*Default Transition* is missing" error. How to connect both states is explained in the following.

A screenshot of the *Stateflow* editor is shown in Figure 3.20. The top part is shared with *Simulink*, while the left part lists all available elements, that can be dragged and dropped into the *Chart*.

The dialog box for possible settings of the *Chart* is shown in Figure 3.21. *Stateflow* supports the use of the two most commonly used types of state machines, *Mealy* and *Moore*. However, it also supports a combination of both, referred to as Classic. In the methodology and applications described in this thesis, however, only *Mealy Chart*s will be used. Therefore the "State Machine Type" is set to "*Mealy*" and further not discussed options are set to recommended values as well. This includes the "Action Language" being set to "C" and activating "Enable C-bit operations". The desktop tools, development environment, and further settings are described in [TM2016i].



(a) Excusive (Or) (b) Parallel (And)

Figure 3.19: *State Decomposition in Stateflow*

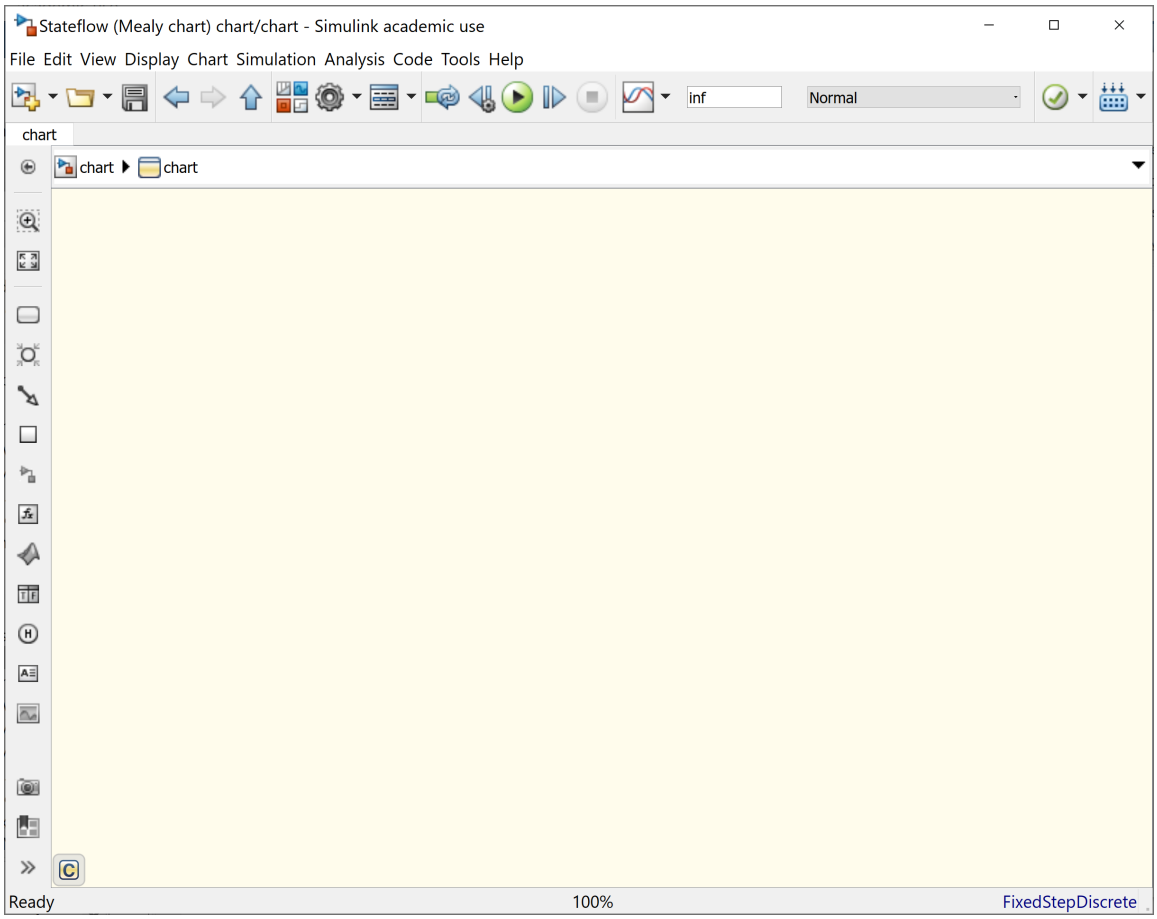


Figure 3.20: *Stateflow Environment*

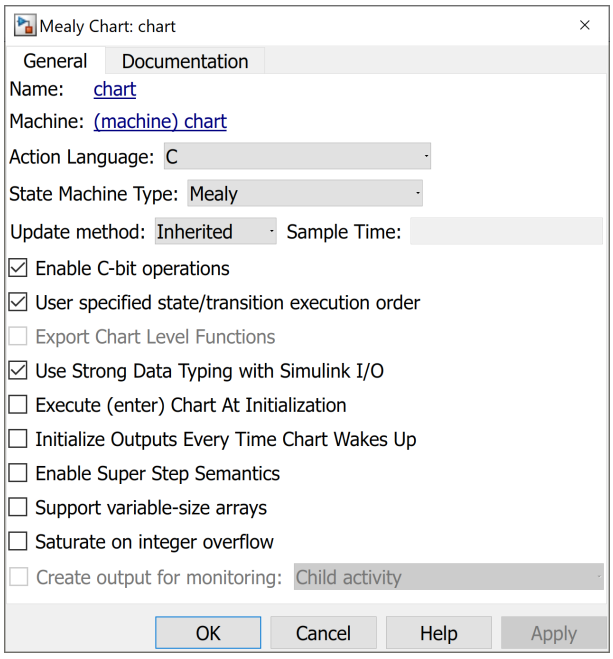


Figure 3.21: *Stateflow Properties*

3.3.2.3 Transition

The *Transition* from one *State* to another is depicted in Figure 3.22. In general, each *Transition* consists of a *Condition*, guarding the *Transition*, and an *Action*, performing some kind of assignment. One exception to this is the *Default Transition*. It initializes the state machines by leading to the initial *State*. An action on the *Default Transition* is referred to as *Default Action* and is often used to initialize the output values. The *Default Transition* can also include conditions, however, there must be one unconditional path to ensure deterministic behavior.

In the example, the *Default Transition* initializes the output to zero and activates *State* `S0`. The *Transition* from *State* `S0` to *State* `S1` is guarded by the *Transition Condition*, which is written in square brackets. Every *Transition Condition* results in a Boolean expression. If this evaluates to `true`, the *Transition* is carried out and the *Transition Action*, in curved brackets, is executed.

Both parts of the *Transition* are connected using a *Junction*. This element is normally used to split a *Transition* or to merge several transitions. However, it can also be used in a very simple way to create easy-to-understand orthogonal connections. The developed guidelines used in the development process at FSD require the *Condition* to be on the horizontal part, while the action must be placed next to the vertical part of the *Transition*.

The *Transition Condition* can contain more than one *Condition*, which need to be connected by using logical operators. The *Transition Actions* can also contain more than one assignment, which in this case need to be separated using a semicolon.

Additionally, *Stateflow* accepts *Condition Actions*, which are placed with a leading "`/`" directly after a condition. In contrast to the *Transition Action*, which is executed when the *Transition* evaluates to `true`, the *Condition Action* is executed when the leading *Condition* is `true`. However, the use of those *Condition Actions* complicates *Charts* without providing a big benefit, which is the reason why they are prohibited by the developed guidelines.

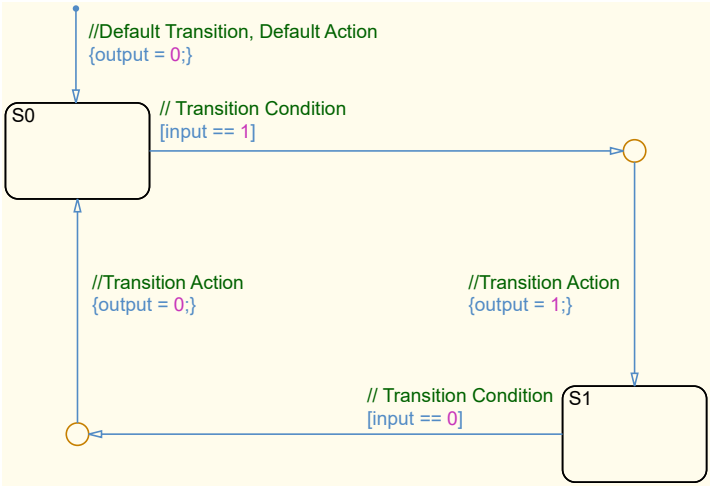


Figure 3.22: *Transitions in Stateflow*

3.3.2.4 Graphical Function

The *Transition Conditions* and *Transition Actions* can be replaced by *Graphical Functions* for increased clarity. In Figure 3.23 the conditions and actions on the *Transition* from `S0` to `S1` as well as on the *Transition* from `S1` to `S0` are replaced with Graphical Functions. Those are located in separate containers at the bottom of the *Chart*.

For *Transition Conditions*, those functions include a unique return value that is mapped to the respective name of the *Graphical Function*. Depending on one or more inputs, this return value can be set within the Graphical Functions and is evaluated on the *Transition* in the *Chart*. For the *Transition Actions*, a simpler version without return value can be used, which sets one or more output values.

In this example, the encapsulation is not very useful, because the conditions and actions only consist of one variable. However, if more complex *Transition Conditions* and *Transition Actions* are used this will improve the readability of the state chart. Additionally, conditions or actions that are used multiple times can easily be replaced by the same *Graphical Function*.

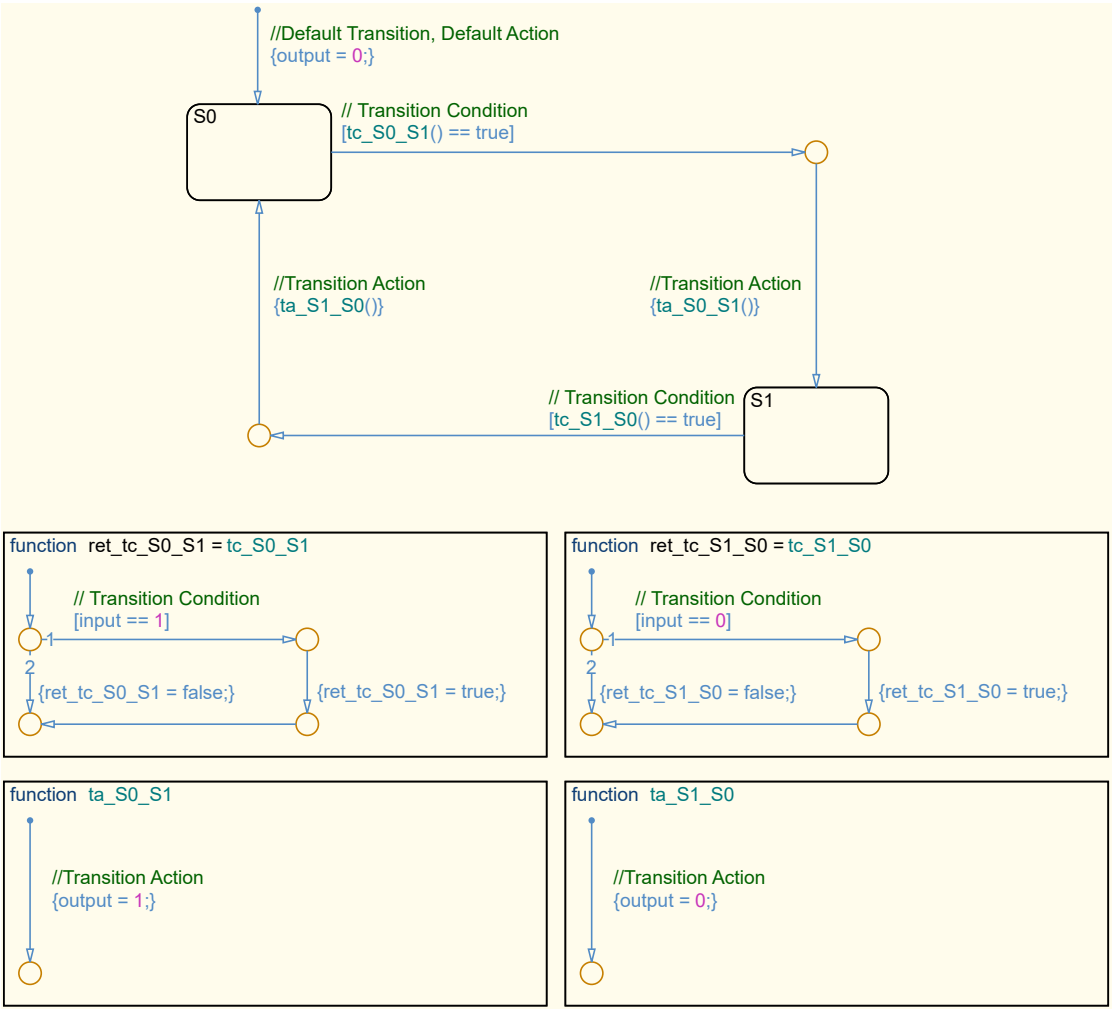


Figure 3.23: *Graphical Functions in Stateflow*

3.3.2.5 Box

In the next step, those Graphical Functions can be placed inside of a *Box*, which is an element to combine multiple functions. Figure 3.24 shows the same state machine as before, with the only addition of boxes. In this thesis boxes in *Stateflow* are used with the "Subchart" option enabled. Therefore, the content is not directly visible in the *Chart*. For this explanation, a representation without this option, e.g. with visible content, is depicted in Figure 3.25 for better understanding.

In both cases, two boxes named "tc", which includes all *Transition Conditions*, and "ta", which includes all *Transition Actions*, are created. In contrast to the *Graphical Functions* the "tc" or "ta" part was removed from the name of the functions since the notation (`BoxName.FunctionName`) already includes this.

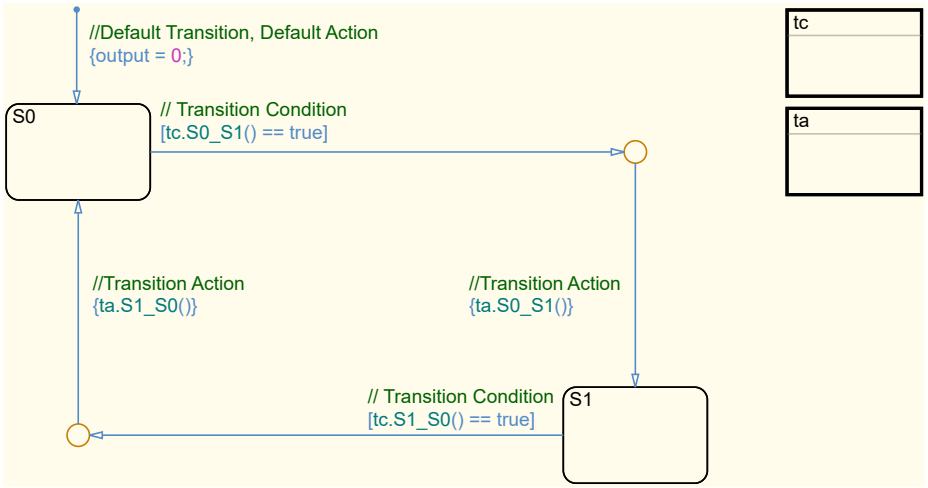


Figure 3.24: *Subchart Boxes in Stateflow*

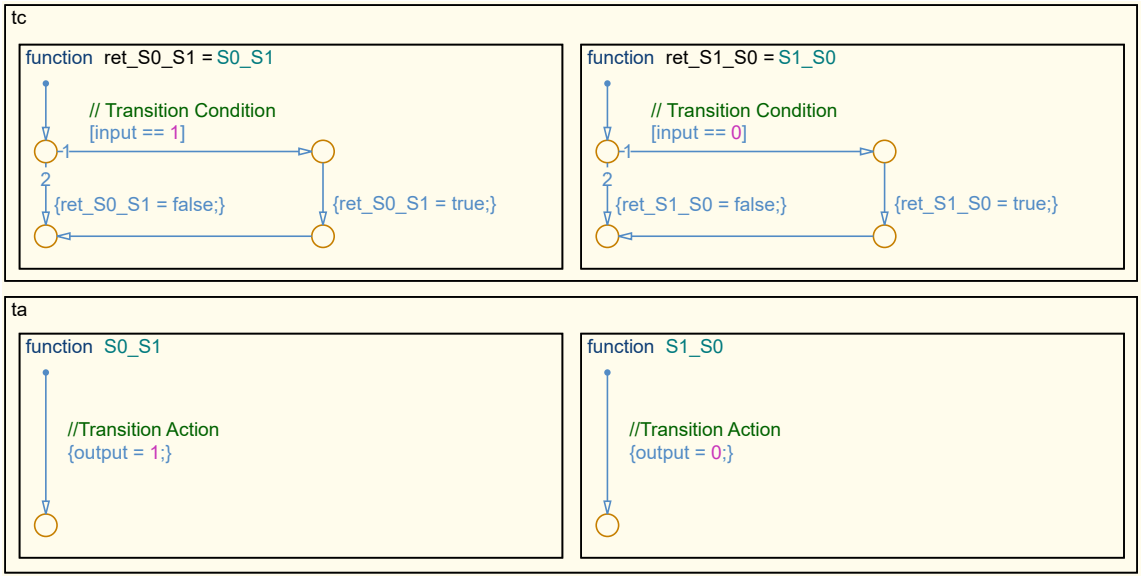


Figure 3.25: *Boxes in Stateflow*

3.3.2.6 Edge Detector

The previously used example of the Edge Detector (c.f. Subsection 3.1.3), implemented as a *Stateflow* chart is depicted in Figure 3.26. It is quite similar to the generic graphical representation in Subsubsection 3.1.3.6. The number of states and transitions are the same, but they are arranged in a different way for better readability.

The default transition is beginning in the top left corner, for easy readability. The next state is selected with respect to the current input. In case the input is "1", `[input==1]`, state `S1` is selected. In all other cases, e.g. if the input is "0", state `S0` is selected. The order of execution is marked with small numbers near the *Junction*. In any case, the output is set to "0".

If in any state the input changes, the respective opposite edge-state is activated and the output is set to "1". In the next step, the output is set to "0" again and the non-edge state is activated. The only exception to this is an immediate additional change of the input. In this case, the respective opposite state is activated, which keeps the output set to "1".

Additionally, the automatically generated code is shown in Listing 3.4. However, it only shows the "step" function (c.f. Subsubsection 3.3.1.5) and automatically generated annotations are removed as well. The complete code is listed in Appendix B. It can be seen that this automatically generated code is much longer than the manually written one in Subsubsection 3.1.3.2. However, optimizations can be applied to the generating process as well as to the compiling process, to reduce performance differences between the two.

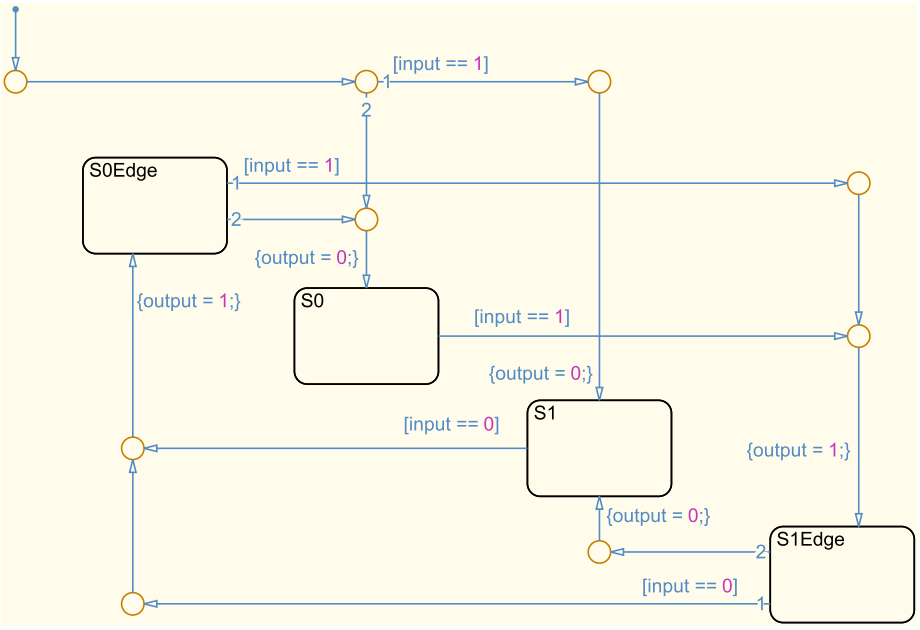


Figure 3.26: *Edge Detector in Stateflow*

```
/* Model step function */
void edgeDetector_step(void)
{
  if (edgeDetector_DW.is_active_c3_edgeDetector == 0U) {
    edgeDetector_DW.is_active_c3_edgeDetector = 1U;
    if (edgeDetector_U.edgeDetector_in == 1) {
      edgeDetector_Y.edgeDetector_out = 0;
      edgeDetector_DW.is_c3_edgeDetector = edgeDetector_IN_S1;
    } else {
      edgeDetector_Y.edgeDetector_out = 0;
      edgeDetector_DW.is_c3_edgeDetector = edgeDetector_IN_S0;
    }
  } else {
    switch (edgeDetector_DW.is_c3_edgeDetector) {
     case edgeDetector_IN_S0:
      if (edgeDetector_U.edgeDetector_in == 1) {
        edgeDetector_Y.edgeDetector_out = 1;
        edgeDetector_DW.is_c3_edgeDetector = edgeDetector_IN_S1Edge;
      }
      break;
     case edgeDetector_IN_S0Edge:
      if (edgeDetector_U.edgeDetector_in == 1) {
        edgeDetector_Y.edgeDetector_out = 1;
        edgeDetector_DW.is_c3_edgeDetector = edgeDetector_IN_S1Edge;
      } else {
        edgeDetector_Y.edgeDetector_out = 0;
        edgeDetector_DW.is_c3_edgeDetector = edgeDetector_IN_S0;
      }
      break;
     case edgeDetector_IN_S1:
      if (edgeDetector_U.edgeDetector_in == 0) {
        edgeDetector_Y.edgeDetector_out = 1;
        edgeDetector_DW.is_c3_edgeDetector = edgeDetector_IN_S0Edge;
      }
      break;
     default:
      if (edgeDetector_U.edgeDetector_in == 0) {
        edgeDetector_Y.edgeDetector_out = 1;
        edgeDetector_DW.is_c3_edgeDetector = edgeDetector_IN_S0Edge;
      } else {
        edgeDetector_Y.edgeDetector_out = 0;
        edgeDetector_DW.is_c3_edgeDetector = edgeDetector_IN_S1;
      }
      break;
    }
  }
}
```

Listing 3.4: *Edge Detector - C-Code generated from Stateflow*

3.3.3 Level Structure

The hierarchical decomposition structure is introduced in Subsection 3.2.5. This subsection will present the explicit implementation using *Simulink* and *Stateflow* and is also associated with contribution *C1.1 - Hierarchical decomposition design strategy, minimizing complexity and optimizing testability*. For this example, an automation with one level and three states is used. For an easier description, additional calculations and encapsulating functions into subsystems are omitted.

Figure 3.27 shows the topmost view of the automation. The `input_bus` is generated in the top left part of the image. For demonstration purposes, it is only filled with constant values, which can be changed during simulation. The `output_bus`, which is generated by the automation, is leaving the subsystem on the right side. However, it is also fed into the subsystem on the lower left part. This is done to generate default values on the `output_bus` because depending on the internal states not all variables of the `output_bus` are propagated in every time step. The `input_bus` consists of `command` and `sensor`, while the `output_bus` consists of `level1_lgx` and `actuator`.

Figure 3.28 depicts the inside of the "level1" subsystem shown in Figure 3.27. For the generic design refer to Figure 3.15. The `input_bus` is entering the subsystem via `input` at input port one and the default values of the `output_bus` via `output_in` at input port two. The `output_bus` is leaving the system via `output` at output port one.

The reddish part in the top left includes the *Stateflow* chart named "level1", which represents the "Level 1 - State Machine". In this example, it has only one input, `command`, and one output, `level1_lgx`. The `command` signal is extracted from the `input_bus` using a "Bus Selector". Therefore, only relevant signals are fed into the state machine. The numerical representation of the calculated state is provided by the chart as an enumerated value. In this case: `state1(1)`, `state2(2)`, or `state3(3)`.

The Input Conditioning and Output Allocation, referenced in Figure 3.15, is represented by the input, output, and "Bus Assignment" in *Simulink*. In this example, additional input verification is omitted.

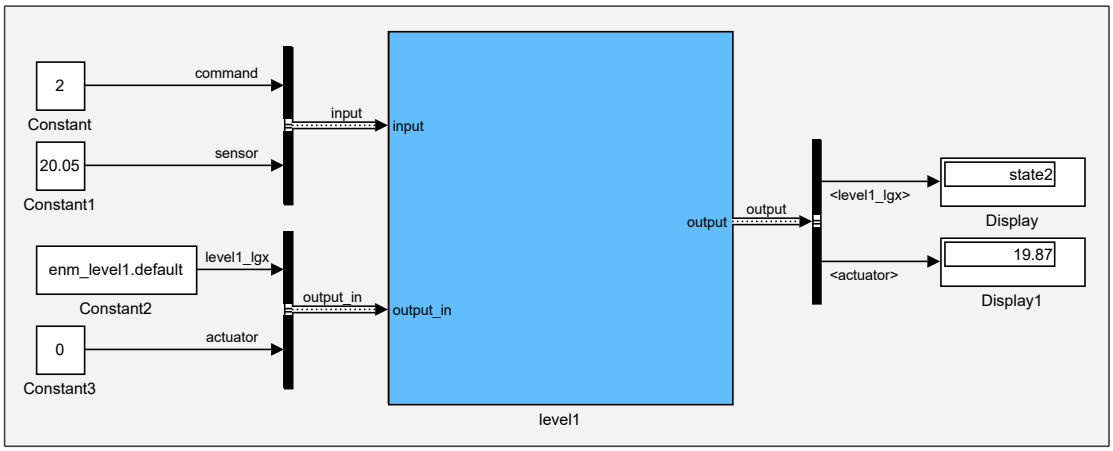


Figure 3.27: *Simulink Scheme*

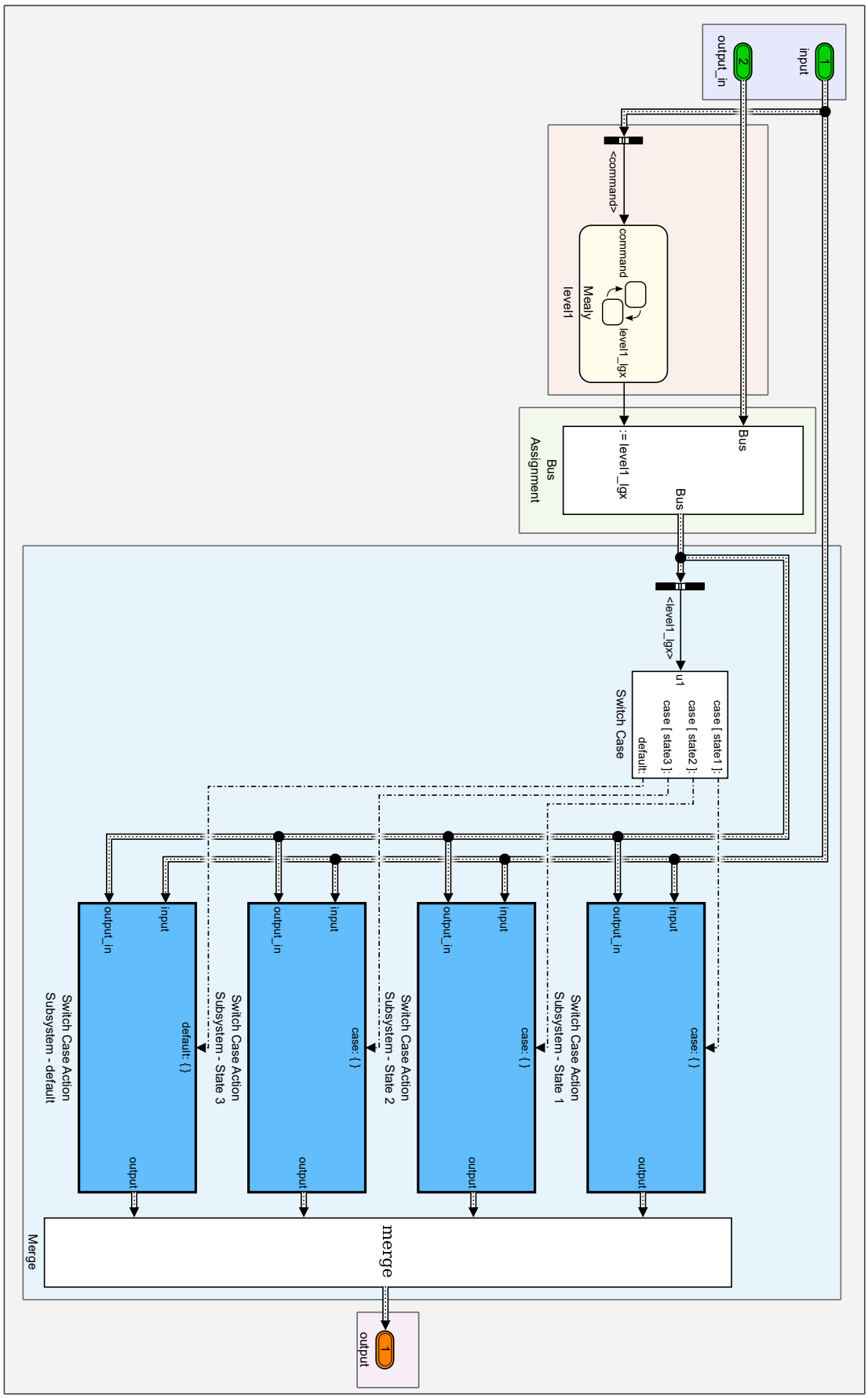


Figure 3.28: *Simulink Scheme - Level 1 Subsystem*

In the greenish part, the `level1_lgx` value of the `output_bus` is assigned to the previously calculated value by the state machine using a "Bus Assignment". Additional calculations can be placed in parallel with the main state machine and their result could also be fed into the `output_bus` in this part, however, in this example, they are omitted.

In the bluish part, on the right side of Figure 3.28, the "Level 1 - Routing" is depicted. This part with two inputs, `input_bus` and `output_bus` (with updated `level1_lgx`), and one output, `output_bus`, is usually encapsulated in an additional subsystem. The routing is using the *Simulink* representation of a "switch-case" construct to create conditionally executed branches. It uses one "Switch Case", multiple "Switch Case Action Subsystems" and a "Merge" block to replicate this functionality. Those three block types need to be used in conjunction to replicate the "switch-case" construct. Due to coding guidelines the "Multiport Switch" block, which has the same functionality, is not used. In the upper left, `level1_lgx` is used to activate the respective subsystem. Each of those uses the same input and output buses as the "Level 1 - Subsystem". The currently active one is then selected by the "Merge" block and passed to the output port.

Please note, an additional default case is used for error mitigation of the switch case construct. The `output_bus` is actually initiated with `default(0)` (c.f. Figure 3.27), representing a state, that the state machine itself cannot represent. Zero is always used as the default state and not used as an enumerated value for the states.

When activating one of those "Switch Case Action Subsystems", which happens in a transition from one state to another, all internal states are reset to their default values. This ensures a deterministic behavior, which is independent of any previous activation. However, if variables from the last execution should be necessary, those need to be stored in external bus elements. While this creates a dependency on external states, in contrast to internal states this is explicitly visible in the implementation.

One of the conditionally executed subsystems is shown in Figure 3.29. In this example, the subsystem only contains a mapping from the `sensor` input to the `actuator` output with an arbitrary gain. In the case of more levels, this subsystem would contain the same structure as depicted in Figure 3.27 and previously shown in Figure 3.15.

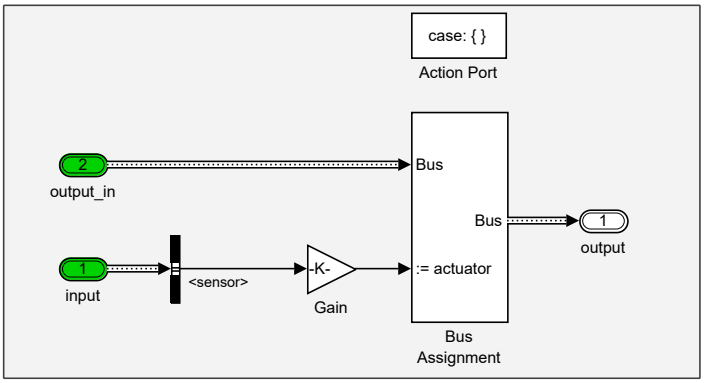


Figure 3.29: *Simulink Subsystem - Switch Case Action Subsystem - State 2*

The state machine used in the example is depicted in Figure 3.30. Its three states are named `S1`, `S2`, and `S3` and arranged in a diagonal pattern from the top left to the bottom right. As indicated by the default transition, the initial state is `S1`. Depending on the `command`, being "1", "2", or "3", the respective state is activated. In the transition actions, an enumerated value is assigned to `level1_lgx`, when entering a state. It can be one of three values, accessed by `enm_level1.state1`, `.state2`, or `.state3`.

In total four enumerations are used to represent the internal state of the *Stateflow* chart. Three actually used values and one default value. A shortened version of their definition is shown in Listing 3.5.

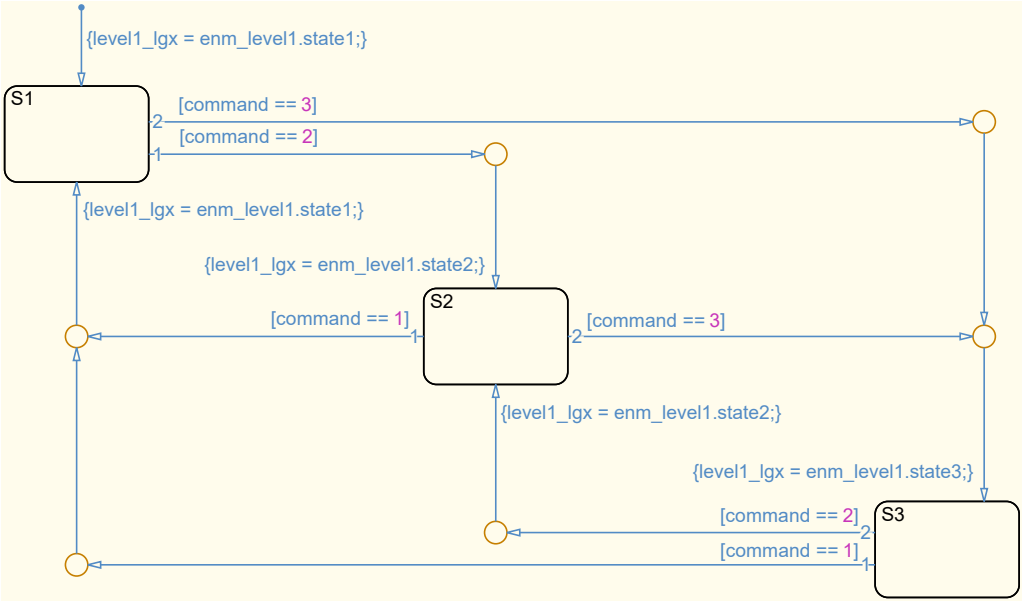


Figure 3.30: *Stateflow - Level 1 - State Machine*

```
classdef enm_level1 < Simulink.IntEnumType
  enumeration
    state_default(0)
    state1(1)
    state2(2)
    state3(3)
  end
  methods(Static = true)
    function retVal = getDefaultValue()
      retVal = enm_level1.default;
    end
    function dScope = getDataScope()
      dScope = 'Exported';
    end
  end
end
```

Listing 3.5: *Simulink - Enumeration Definition*

3.3.4 Modeling Guidelines

This subsection is associated with contribution *C1.2 - Modeling guidelines for implementation, minimizing opacity and maximizing software maintainability.* It provides an overview of developed guidelines that are relevant to the design, implementation, and testing of state machines. Some have been derived from and are an addition to the *Control Algorithm Modeling Guidelines using MATLAB Simulink, and Stateflow* [Mat2012, Mat2016], the *Simulink - Modeling Guidelines for Code Generation* [TM2016u], the *Simulink - Modeling Guidelines for High-Integrity Systems* [TM2016t], and the *Mathworks Model Architecture Guidelines* [TM2016m, TM2016n, TM2016o]. Additionally, the block set and functionality are restricted to provide full compatibility with the *Simulink Code Inspector* [TM2016r].

In this context, the developed software modules are used as design models and replace the normal software requirements [TM2016f]. This is done in accordance with the aviation software development guidelines RTCA DO-178B [RTC1992], RTCA DO-178C [RTC2011a], and RTCA DO-331 [RTC2011b].

Based on MathWorks Automotive Advisory Board (MAAB), the modeling guidelines are listed with their four main attributes, as shown in the following [Mat2012, p. 7ff].

- Title
- Scope
- Description
- Rationale
  - Readability, Workflow, Simulation and Analysis, Verification, Code Generation

The title is a short unique name, highlighting the main content of the guideline, while the scope describes the field of application within *Simulink*, *Stateflow*, the state machine, or the state chart. The description of each guideline in the table gives a short overview, while further explanations are given in the text, where necessary. The rationale describes the reason why the guideline is recommended.

It can be one or more of the following: readability, workflow, simulation and analysis, verification, or code generation. Readability refers to easy-to-understand algorithms, uniform model appearance, and clean interfaces. Workflow describes an effective development process and workflow, model maintenance, changes, or portability. Simulation and analysis guidelines target efficient simulation and analysis with respect to speed, memory use, or model instrumentation. Verification refers to requirement traceability, testing, and test integration. Code generation targets fast software changes and code generation robustness. [Mat2012, p. 10]

Further recommended guideline fields like an identification number, priority, *MATLAB* version, prerequisites, and last change are omitted in the following.

An overview of the developed modeling guidelines, which are discussed in the following, is given in Table 3.9. Discussion on some guidelines is shortened since their concept has been introduced previously.

The state chart settings scope refers to general adjustments within the *Stateflow* chart itself like the type of state machine, used programming language, and scope of variables. The state chart interface scope is targeted at the interface between *Simulink* and *Stateflow*. The scope of state machine elements describes which types of *Stateflow* charts are used and which elements are used within those, to create the required logic. The general setup and positioning are described in the state machine structure scope. The state machine modeling scope consists of general objectives of *Stateflow*, the replacement of available features with more advanced solutions, and concatenation of multiple conditions. The more specific placement of states and transitions in *Stateflow* itself is included in the *Stateflow* arrangement scope. Signal naming and routing are covered in the *Simulink* structure scope.

Table 3.9: *Modeling Guidelines - Overview*

Scope	**Title**
State Chart - Settings	State Machine Type
	Action Language
	Data Scope
State Chart - Interface	State Chart Interfaces
	State Chart Events
	Enumerated State Information
State Machine - Elements	Stateflow Elements
	State Chart Elements
	State Chart Containers
State Machine - Structure	State Decomposition
	Self-Recurring States
	Condition Actions
State Machine - Modeling	Stateflow Objectives
	Temporal Logic
	Logical Sequencing
Stateflow - Arrangement	State Arrangement
	Transition Arrangement
	Conditions and Action Arrangement
Simulink - Structure	Signal Naming
	Signale Routing

3.3.4.1 State Chart - Settings

The guidelines attributed to the settings of the state chart are listed in Table 3.10, Table 3.11, and Table 3.12. Various other settings of *MATLAB*, *Simulink*, and *Stateflow* are omitted here since they are part of the general model development process at FSD.

As introduced in Subsubsection 3.3.2.2, state charts in *Stateflow* can be set to *Mealy*, *Moore*, or *Classic*, the latter being a combination of the first two. To generate consistent state machines, *Mealy* must be selected. Reasons include faster reaction to inputs and encapsulation of decision logic into flow charts. Additionally, the Simulink Code Inspector (SlCI) can be used to support code-review objectives when using *Mealy* charts.

The action language, expressions used for transitions in state flow, can be set to *MATLAB* or "C". For the development process, "C" is chosen since it enforces stronger data typing, scalar operations, and less inherited properties. [TM2016e, TM2016b]

The data scope in a chart in *Stateflow* can be set to various types, to define the impact range of the variables. However, it shall be limited to *input*, *output*, and *parameter* due to clarity and readability. Additionally, *parameter* should be avoided wherever possible.

Table 3.10: *Guidelines - State Machine Type*

Title	State Machine Type
Scope	State Chart - Settings
Description	*Mealy* shall be used as state machine type. This can be selected in Chart/Properties. Moore or Classis must not be used to avoid conflicting implementations.
Rationale	Readability, Workflow, Verification

Table 3.11: *Guidelines - Action Language*

Title	Action Language
Scope	State Chart - Settings
Description	Action language shall be set to C in chart properties.
Rationale	Readability, Workflow, Code generation

Table 3.12: *Guidelines - Data Scope*

Title	Data Scope
Scope	State Chart - Settings
Description	Data scope in *Stateflow* shall be limited to *input*, *output*, and *parameter*. Other data scopes: *local*, *constant*, and *data store memory* shall not be used. Additionally, *parameter* shall be avoided whenever possible.
Rationale	Readability, Workflow

3.3.4.2 State Chart - Interface

The guidelines attributed to the interface of the state chart are listed in Table 3.13, Table 3.14, and Table 3.15. They address the types of variables allowed to go to or from a state machine.

Only scalar elements are allowed as input and output to a state machine. Therefore, all other types like vectors, matrices, and buses shall not be used. The main reason being explicit visibility of all necessary variables and reduced complexity within the state chart.

Stateflow also supports the use of events, which can be used to trigger the next evaluation cycle in state charts or to trigger action subsystems in *Simulink*. However, this feature is generally not necessary and can easily be replaced.

The developed hierarchical decomposition structure makes extensive use of separating large and complex state machines into small and less complex ones. This means state information needs to be passed throughout various levels of the automation. Using enumerated data types resolves interpretation errors and improves readability.

Table 3.13: *Guidelines - State Chart Interfaces*

Title	State Chart Interfaces
Scope	State Chart - Interface
Description	Signals traversing borders of a state chart shall be limited to those directly necessary within this state chart. Signals from or to multi-signal containers like buses shall be selected outside of the state machine.
Rationale	Readability, Workflow, Verification

Table 3.14: *Guidelines - State Chart Events*

Title	State Chart Events
Scope	State Chart - Interface
Description	Events shall not be used as input or output to *Stateflow*. Functionality may be replaced, if absolutely necessary, by triggered subsystems in *Simulink*.
Rationale	Readability, Workflow

Table 3.15: *Guidelines - Enumerated State Information*

Title	Enumerated State Information
Scope	State Chart - Interface
Description	Enumerated data types shall be used within one multi-level state machine to pass active state information through all levels.
Rationale	Readability, Workflow

3.3.4.3 State Machine - Elements

The guidelines attributed to the elements of the state chart are listed in Table 3.16, Table 3.17, and Table 3.18. They address the available elements in *Stateflow*.

Stateflow supports multiple base elements that can be used to create state machines. Besides the *Chart*, which is a graphical representation of the state machines it also supports a *State Transition Table* and a *Truth Table*. However, the latter two are considered as different modeling languages and would require additional checks and documentation.

The *Stateflow* Charts, in turn, support also various elements to design and implement state machines. Those elements are restricted to *State*, *Junction*, *(Default) Transition*, *Box*, *Graphical Function*, and *Annotation*. This leads to consistently implemented state machines with improved readability.

Additionally, enhanced containers like *Group*, *Subchart*, and *Atomic Subchart* shall not be used. Due to the hierarchical decomposition structure, those elements are no longer necessary and would lead to complex state machines.

Table 3.16: *Guidelines - Stateflow Elements*

Title	Stateflow Elements
Scope	State Machine - Elements
Description	The only element from the *Stateflow* toolbox that shall be used is the *Chart*. The *State Transition Table* and *Truth Table* shall not be used.
Rationale	Readability, Workflow, Verification, Code generation

Table 3.17: *Guidelines - State Chart Elements*

Title	State Chart Elements
Scope	State Machine - Elements
Description	The following state chart elements can be used: *State*, *Junction*, *(Default) Transition*, *Box*, *Graphical Function*, and *Annotation*. The following state chart element shall not be used: *Simulink Function*, *MATLAB Function*, *Truth Table*, *History*, *Image*.
Rationale	Readability, Workflow, Verification, Code generation

Table 3.18: *Guidelines - State Chart Containers*

Title	State Chart Containers
Scope	State Machine - Elements
Description	The following *Stateflow* containers shall not be used: *Group*, *Subchart*, *Atomic Subchart*. Subcharts for boxes being the only exception.
Rationale	Readability, Workflow, Verification, Code generation

3.3.4.4 State Machine - Structure

The guidelines attributed to the structure of the state chart are listed in Table 3.19, Table 3.20, and Table 3.21. They address the general structure of allowed states and transitions within a state chart.

Stateflow allows for two types of state decomposition settings, *OR* (Exclusive) and *AND* (Parallel). The setting applies to all states within a chart and within this methodology, only exclusive states are used.

Transitions generally connect two states to realize a change if some input requirements are met. However, transitions can also lead back to the state they were coming from. This could be done to e.g. count time steps in a given state. However to reduce complexity transitions leading to the originating state are forbidden.

In *Stateflow*, actions in a transition can either be executed if a certain condition along the transition is met or if the complete transition to another state is conducted. The guidelines, however, prohibit the use of the so-called *Conditions Actions*.

Table 3.19: *Guidelines - State Decomposition*

Title	State Decomposition
Scope	State Machine - Structure
Description	Only *OR* (Exclusive) state decomposition shall be used. *AND* (Parallel) decomposition, where more than one state can be active at the same time, is forbidden.
Rationale	Readability, Workflow

Table 3.20: *Guidelines - Self-Recurring States*

Title	Self-Recurring States
Scope	State Machine - Structure
Description	Self-recurring states shall not be used. Therefore transitions that change an output but return to the same state are not allowed.
Rationale	Readability, Workflow, Verification, Code generation

Table 3.21: *Guidelines - Condition Actions*

Title	Condition Actions
Scope	State Machine - Structure
Description	Only transition actions, which change an output after a successful transition, shall be used. *Condition Actions*, which are actions that can change an output even if the transition is not completed, are forbidden.
Rationale	Readability, Workflow, Code generation

3.3.4.5 State Machine - Modeling

The guidelines attributed to the modeling of state charts are listed in Table 3.22, Table 3.23, and Table 3.24, which address generic modeling characteristics of state machines.

Due to the combination of *Stateflow*, and *Simulink* the purpose of the state machines can be restricted to mode switching only. Any additional mathematical computations shall be performed outside of the individual state chart. This leads to a clear separation of assigned tasks with explicit inputs and outputs.

The use of an explicitly implemented external counter is introduced in Subsubsection 3.2.4.3. Therefore, the temporal logic, integrated into *Stateflow*, shall not be used.

Logical Sequencing on the state chart level shall be performed using logical operators to generate one logical expression per transition. However, in encapsulated graphical functions, graphical sequencing can be used if necessary, for increased readability.

Table 3.22: *Guidelines - Stateflow Objectives*

Title	Stateflow Objectives
Scope	State Machine - Modeling
Description	The purpose of *Stateflow* shall be restricted to mode switching. Mathematical computations shall be performed in conditionally executed *Simulink* subsystems.
Rationale	Readability, Workflow, Simulation and Analysis, Verification

Table 3.23: *Guidelines - Temporal Logic*

Title	Temporal Logic
Scope	State Machine - Modeling
Description	*Stateflow* temporal logic shall not be used. To replace this functionality an external timer shall be used that can be, if necessary, reset from the state chart.
Rationale	Readability, Simulation and Analysis, Verification, Code generation

Table 3.24: *Guidelines - Logical Sequencing*

Title	Logical Sequencing
Scope	State Machine - Modeling
Description	Logical sequencing shall be performed using logical operators in the state chart. In encapsulated *Graphical Functions*, graphical sequencing can be used if favorable for the illustration.
Rationale	Readability, Workflow

3.3.4.6 Stateflow - Arrangement

The guidelines attributed to the arrangement of elements in the state chart are listed in Table 3.25, Table 3.26, and Table 3.27.

They address the arrangement of states, transitions, conditions, and actions in the chart, and all target better readability and easier understanding of the state charts. An example where all three guidelines are applied is the Edge Detector depicted in Figure 3.26.

Table 3.25: *Guidelines - State Arrangement*

Title	State Arrangement
Scope	Stateflow - Arrangement
Description	All states within *Stateflow* shall be arranged diagonally starting from top left to bottom right. The initial state, if only one exists, shall be placed in the top left.
Rationale	Readability, Workflow

Table 3.26: *Guidelines - Transition Arrangement*

Title	Transition Arrangement
Scope	Stateflow - Arrangement
Description	The transitions between all states shall be orthogonal by using *Junctions*. Transitions leading from a higher state to a lower state shall be placed on the top right half of the state chart with a minimum number of crossings. Likewise, transitions from lower to higher states shall be placed on the lower-left half of the state chart.
Rationale	Readability, Workflow

Table 3.27: *Guidelines - Condition and Action Arrangement*

Title	Conditions and Action Arrangement
Scope	Stateflow - Arrangement
Description	Conditions shall be placed above the respective transition on the horizontal part. Actions shall be placed on the inside, with respect to the state chart, of the transitions on the vertical part. Consolidation of multiple transitions is allowed if conditions are placed directly at the output of the individual states.
Rationale	Readability, Workflow

3.3.4.7 Simulink - Structure

The guidelines attributed to the hierarchical decomposition structure using *Simulink* and *Stateflow* are listed in Table 3.28 and Table 3.29. They address naming conventions for an easier understanding of state machines and their interactions as well as the specific realization in *Simulink*.

Table 3.28: *Guidelines - Signal Naming*

Title	Signal Naming
Scope	Simulink - Structure
Description	The following postfixes for variables shall be used within the automation modules. • A `_flg` means flag, represents a boolean variable and is used for triggering an event when set to `true` • A `_cfg` means change flag, represents a boolean variable and is used for triggering an event on both edges • A `_rfg` means rising flag, represents a boolean variable and is used for triggering an event on the rising edge • A `_ffg` means falling flag, represents a boolean variable and is used for triggering an event on the falling edge • A `_idx` means index, represents an unsigned integer variable and is used for generic indices • A `_lgx` means logic, represents an unsigned integer variable and is used for defining enumerated values or states
Rationale	Readability, Workflow

Table 3.29: *Guidelines - Signal Routing*

Title	Signal Routing
Scope	Simulink - Structure
Description	The multi-level state machines shall use one input bus containing all necessary external information and one output bus containing all generated information and commands that are relevant for other modules. The *Bus Assignment* shall be used for signal consolidation. For selecting the next lower active mode a construct of *Switch Case*, *Switch Case Action Subsystem*, and *Merge* shall be used.
Rationale	Readability, Workflow, Simulation and Analysis, Verification

3.4 Testing and Verification

After the last two sections, Section 3.2 and Section 3.3, described the design and implementation aspects of this methodology, this section presents the testing and verification activities. The different steps or test environments include Unit Tests, Model Checking using formal methods, MiL, SiL, HiL, AiL, Ground Tests, and Flight Tests. Those verification methods target different systems, or parts of the system, under test which are the software function, design model, C-Code, FCC, actuators, and sensors. An overview of the testing and verification steps, which are part of this methodology and are applied to the automation functions presented in this thesis, is depicted in Figure 3.31.

Contribution *C1.3 - Incremental bottom-up application of formal methods, ensuring effective testing and guaranteed system characteristics* is presented in Subsection 3.4.2.

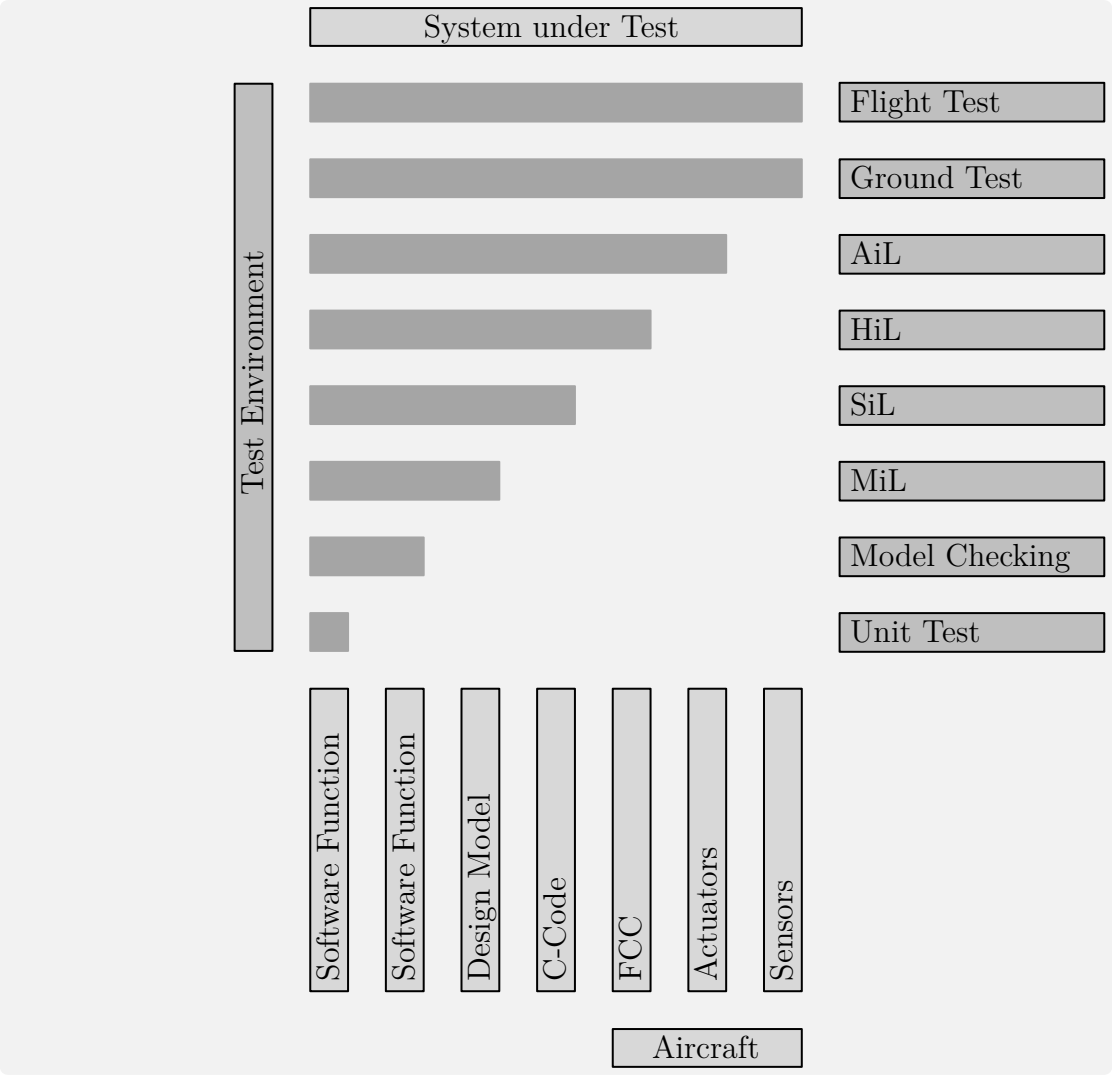


Figure 3.31: *Testing Overview*

3.4.1 Unit Tests

The *MATLAB Unit Testing Framework* is used to define, simulate, and analyze test cases. Although primarily built for *MATLAB*, *Simulink*, as well as *Stateflow* models, can be executed and therefore integrated into those tests. [TM2014]

An overview of the *MATLAB* Unit Tests and their functions is presented in Listing 3.6. Those include the main function (`unitTests`), `setupOnce`, `setup`, the test itself (`test_unitTest1`), `teardown`, and `teardownOnce`. The objective of the respective function is summarized in the comments of Listing 3.6.

The main function, named `unitTests`, is used to create a suite of one or more tests. Those tests are defined as local functions and have to start with "`test_`" in order to be recognized by the subfunction `functiontests`. The function `setupOnce` is only called once in the beginning. It can be used to initialize the simulation model or to set other variables that are not dependent on the individual test case. In contrast to that function, the function `setup` is called before each individual test and can be used to reset variables, which might have been used by previous tests to allow for a defined state of the model. Each test case is wrapped in an individual function, which in this example is `test_unitTest1`. Within this function, inputs for the tests are defined and the test is executed. Similar to the pre-test functions, `teardown` and `teardownOnce` are called after completion of one test or all tests respectively. They are used to extract and save data from the test case or to perform a generic cleanup after all test cases.

In this methodology Unit Tests are used to perform tests of the individual state machines, the combination of all state machines into one automation, and even for interaction tests of the automation with the rest of the flight control algorithm.

```
function tests = unitTests %Main function
    tests = functiontests(localfunctions); %Create test suite
  end
function setupOnce(testCase)
    %Initialize model
  end
function setup(testCase)
    %Reset variables
  end
function test_unitTest1(testCase)
    %Define test inputs, Run test case
  end
function teardown(testCase)
    %Extract Data, Save Test Data
  end
function teardownOnce(testCase)
    %Cleanup
  end
```

Listing 3.6: *MATLAB - Unit Tests*

3.4.2 Model Checking

Model Checking is an automatic formal verification technique, in which logic specifications and a model of the system are used in combination with a search algorithm to determine whether or not the state machine satisfies the requirements. [Cla1997, p. 54]

Formal methods are used within Model Checking to verify complex software designs. It is an alternative to other verification approaches like simulation and testing and has been studied since the emerge of the first computers in the 1960s. [MH2003, Roz2011]

While simulation and testing are most effective in the early stages of development and debugging, the effectiveness of model checking does not depend on the development stage. This can be explained by the number of errors in software. While it is comparatively easy to find them at the beginning of the design and development process it gets increasingly harder to find the errors using simulation and testing as the development becomes cleaner and better. The biggest disadvantage of those techniques is the incapability to draw any conclusion on remaining errors in the software. [CGP1999, p. 1ff]

One example of a major software error, that was not found during simulation and testing, resulted in the loss of a rocket. On June 4, 1996, the Ariane 5 Flight 501 lasted less than a minute. The error was located in the software of the Inertial Reference System (IRS) that calculates the attitude and movement of the rocket. The failure was caused by an unprotected data conversion from a 64-bit floating-point to a 16-bit signed integer. This resulted in the abrupt movement, which led to a rupture between the core stage and the boosters, which in turn correctly triggered the self-destruction. [Lio1996]

The verification of the software using model checking could probably have prevented this problem. Another advantage of model checking is the ability to generate counterexamples when a given property is falsified. This can be used to identify the erroneous part of the software or to rewrite the test case that had incorrect properties.

However, the state explosion problem is the main challenge in model checking. It refers to systems with many states and/or input signals with a large range. In this case, the combinatorial complexity grows "exponentially". Even though solutions, like replacing Binary Decision Diagrams (BDDs) with Boolean Decision Procedures (BDPs) [BCCZ1999], can cope with this disadvantage of model checking it is best to avoid the problem by decreasing the number of states and range of input signals. This methodology is built around the concept of minimizing the complexity of individual state machines.

Contribution *C1.3 - Incremental bottom-up application of formal methods, ensuring effective testing and guaranteed system characteristics* is developed to verify the correct functionality of the individual state machines as well as the overall operation of the flight control automation. A propagation of input and output ranges, across all levels, is used to guarantee that those variables are bounded.

The tool that is extensively used to perform model checking in *Simulink* and *Stateflow*, the Simulink Design Verifier (SDV), is explained in the following. It has various modes and functions and can even be used to solve mathematical games like Sudoku [Sto2010].

3.4.2.1 Simulink Design Verifier

The Simulink Design Verifier (SDV) is used to identify and isolate design errors, generate tests, and prove properties. It applies Model Checking via formal methods to perform *Design Error Detection*, *Test Generation*, and *Property Proving*. [TM2016s, p. 1-2]

The *Design Error Detection* can be used to detect flaws like dead logic, integer overflow, and division by zero. In case of a detected violation, a test case provoking the violation is automatically generated for debugging. Additionally, the SDV can be used for *Test Generation* to achieve various objectives. This includes coverage requirements like condition coverage or decision coverage and even Modified Condition / Decision Coverage (MC/DC) [TM2019]. Furthermore, the SDV is used for *Property Proving* of custom requirements, meaning that the tool can guarantee that a property holds true under given input ranges or alternatively provide a counterexample in case of violation.

Contribution *C1.3 - Incremental bottom-up application of formal methods, ensuring effective testing and guaranteed system characteristics*, makes extensive use of the SDV. Due to the smaller scope, ensured by the design principles introduced in Subsection 3.2.2, the Model Checking process using formal methods can be conducted more efficiently. Part of the contribution is a *Simulink* model environment, which is used to automatically perform design error detection, test case coverage analysis, and property proving of key functionalities of all state machines. Additionally, a test report is generated for all state machines and all tests, using the developed *MATLAB* script. The developed code for this functionality is available in Appendix C. Furthermore, an incremental bottom-up application of formal methods is developed.

To create a seamlessly tested system with guaranteed system characteristics the SDV is used throughout all levels of the developed applications. On each level, starting from the lowest, assumptions are made about the inputs, which are used to guarantee certain outputs, that match the requirements. The assumptions are mostly about ranges, but can also include a possible sequence of inputs or a possible combination of inputs. On the next level, those assumptions from the lower level are then used as requirements for the outputs on the higher level. Using formal methods by applying different modes of the SDV, those are verified with the new assumptions about the inputs on this level. This procedure can be used all the way to the top level, where the assumptions of the inputs need to be matched by input requirements for the system.

The SDV verifies or falsifies certain objectives. In the following, they will be listed in tables and marked either green or red depending on the outcome of the respective objective. They are numbered in ascending order, however, due to the internal structure of the SDV, some are not used and therefore also not listed in the tables.

When automatically generating an example, the SDV produces a test case that is specified in the report in a table format and also used in the following. This table shows the simulation step, starting from one, and the simulation time, starting from zero. Since $10ms$ is used as step size for the following examples, step two corresponds to $0.01s$.

The SDV is used in this methodology to perform model checking and is applied to the flight control automation functions presented in this thesis. Its three main use cases are summarized below and presented in more detail in the following.

- Design Error Detection
  - Dead Logic, Identify Active Logic
  - Integer Overflow, Division by Zero,
    (Check specified intermediate minimum and maximum values,
    Out of bound array access)
- Test Generation, Model Coverage Objectives: MCDC
- Property Proving

3.4.2.2 Design Error Detection

If the SDV is run in Design Error Detection mode, it can detect the following types of errors: dead logic, integer or fixed-point data overflow, division by zero, intermediate signal values that are outside of the specified minimum and maximum values, and out of bound array access.

However, when in this mode, the SDV can either be used to detect dead logic or integer overflow and the other objectives, which is also shown in the overview in Subsubsection 3.4.2.1. This is due to the fact, that the SDV uses different engines to check for those errors. Polyspace Code Prover uses semantic analysis and abstract interpretation based on formal methods to prove the absence of run-time errors [TM2016q], while Polyspace Bugfinder uses static analysis to identify software bugs [TM2016p].

The test model that is used in the following to demonstrate the functionality of the SDV is depicted in Figure 3.32. It consists of two inputs and two outputs, however depending on the test case used in the following, different erroneous implementations, and therefore not necessarily all interfaces are used.

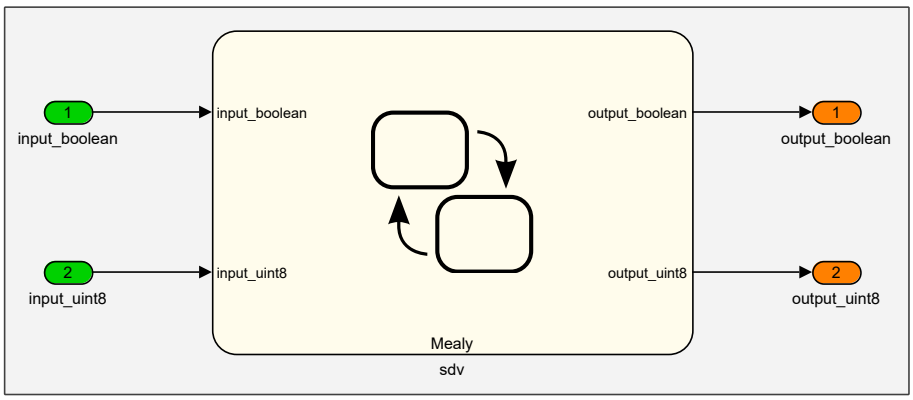


Figure 3.32: *Simulink Design Verifier - Test Model*

3.4.2.3 Design Error Detection - Dead Logic

The SDV in Design Error Detection mode with activated settings for dead logic and active logic will check the current design, in this case, a *Stateflow* chart, for any states and/or transitions that cannot be reached. This test incorporates the input signals and identifies the reachable or unreachable part of the model depending on their range.

In the following example, only one 8-bit unsigned input and three states are used. The resulting visual representation is shown in Figure 3.33, while a slightly modified objectives table is shown in Table 3.30. The analysis of the SDV includes nine objectives. One for each state and two (true and false) for each decision. It can be seen that transition with execution order two from state `S0` will not be used. This turns both the transition condition, as well as state `S1` into dead logic. Additionally, the transition from state `S2` to `S0` will always result in `true`, which is also identified as dead logic.

Table 3.30: *Design Error Detection - Objectives Status*

#	Model Item	Description	Logic
1	sldv_ded_dl	Chart: Substate executed State "S0"	Active
2	sldv_ded_dl	Chart: Substate executed State "S1"	Dead
3	sldv_ded_dl	Chart: Substate executed State "S2"	Active
4	[input_uint8 >0]	Transition: Transition trigger expression F	Active
5	[input_uint8 >0]	Transition: Transition trigger expression T	Active
6	[input_uint8 >= 0]	Transition: Transition trigger expression F	Dead
7	[input_uint8 >= 0]	Transition: Transition trigger expression T	Active
9	[input_uint8 >0]	Transition: Transition trigger expression F	Active
10	[input_uint8 >0]	Transition: Transition trigger expression T	Dead

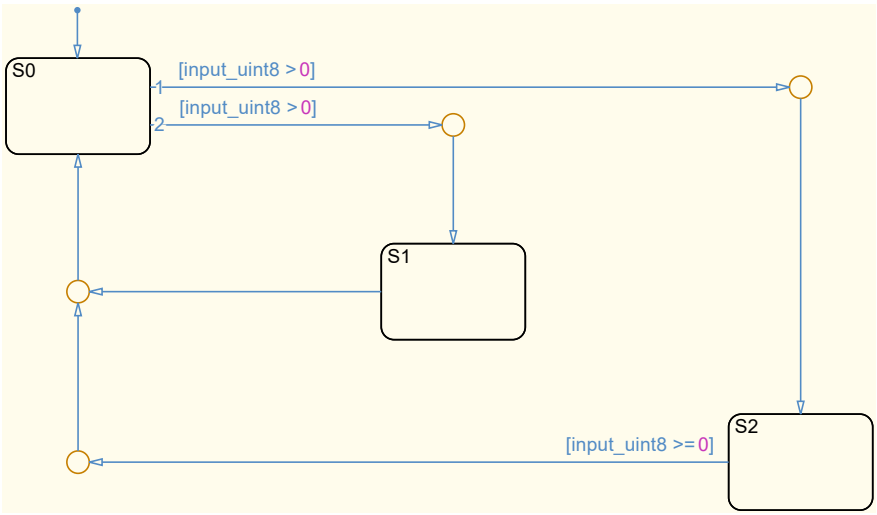


Figure 3.33: *Design Error Detection - Dead Logic*

3.4.2.4 Design Error Detection - Integer Overflow, Division by Zero

The SDV in Design Error Detection mode, but settings for integer overflow and division by zero enabled can be used to identify other types of errors in the model. Figure 3.34 depicts the example *Stateflow* chart analyzed in this mode, with its implementation errors.

This mode of the SDV focuses only on the outputs, as depicted by the objectives in Table 3.31. An overflow objective is analyzed for every assignment. However, division by zero is only analyzed in actions with divisions, for obvious reasons. In case the SDV has failed to reach the objective a counterexample, listed in Table 3.32, is generated.

Objective five, which is analyzing the overflow by addition of the transition action leading to state `S1`, is falsified by test case one. An input greater than zero is necessary at step two to active the transition leading to state `S1`. In this case, an input of 255 leads to an overflow of the output, since the addition of one cannot be represented within an 8-bit variable.

In a similar way, objective three is falsified by test case two, which uses an input of zero in step two. This activates the transition from state `S0` to state `S2` and therefore causes a division by zero.

Table 3.31: *Design Error Detection - Objectives Status*

#	Model Item	Description	Test Case
3	"{output_uint8= 10/input_uint8"	Division by zero: /	2
4	"{output_uint8= 10/input_uint8"	Overflow: /	n/a
5	"{output_uint8= input_uint8+1}"	Overflow: +	1

Table 3.32: *Design Error Detection - Test Cases*

Time	0.00	0.01
Step	1	2
Test Case 1, Input: input_uint8	1	255
Test Case 2, Input: input_uint8	128	0

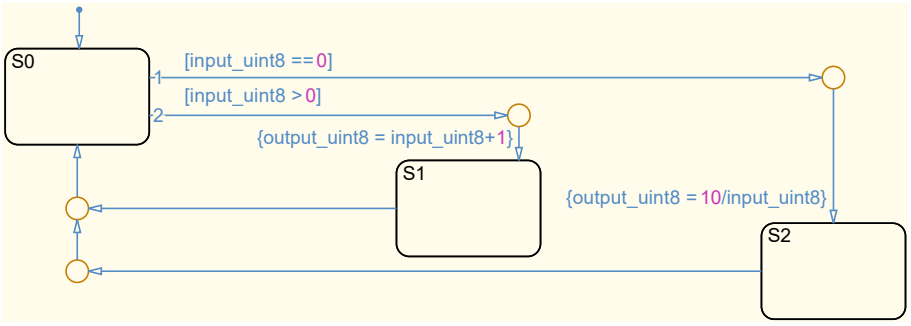


Figure 3.34: *Design Error Detection - Integer Overflow, Division by Zero*

3.4.2.5 Test Generation

The SDV can also be used in test generation mode with model coverage objectives: condition coverage, decision coverage, and MC/DC.

MC/DC is a code coverage requirement, which is used for testing and evaluating of safety-critical software that requires the criteria listed in the following [HVCR2001, p. 7].

- Every point of entry and exit has been invoked at least once
- Every decision has taken all possible outcomes at least once
- Every condition in a decision has taken all possible outcomes at least once
- Every condition in a decision has been shown to independently affect the outcome

Although model coverage analyses are not required for the methodology and applications described in this thesis, it is used throughout the design, implementation, and testing of the software to reduce the error probability. An example model for a test generation with MC/DC analysis is shown in Figure 3.35. The respective number of objectives and archived coverage, for this model with intentional errors, is listed in Table 3.33.

Table 3.33: *Test Generation - Objectives Overview*

	Decision	**Condition**	**MCDC**
Objectives	15	8	8
Coverage	87%	88%	75%

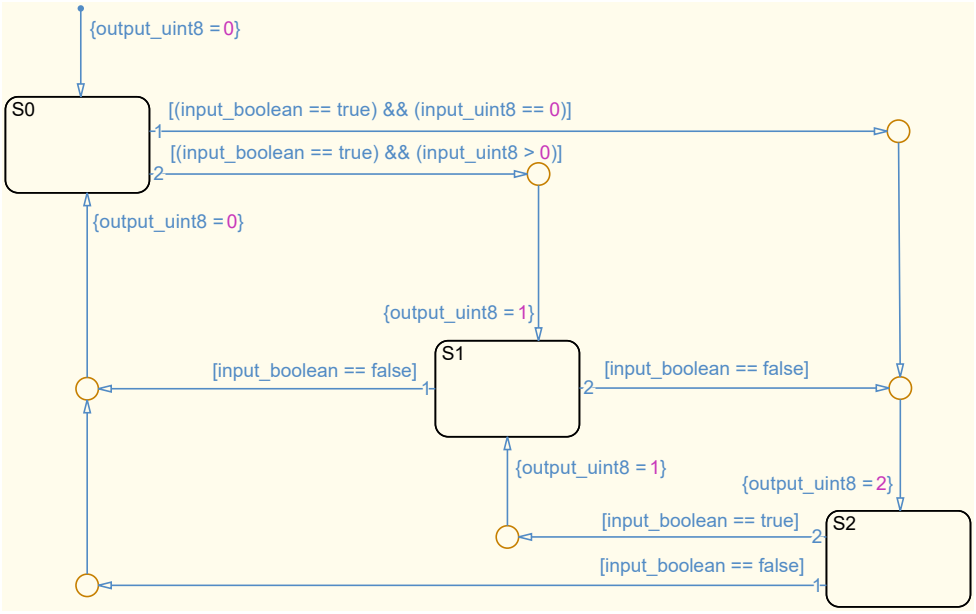


Figure 3.35: *Test Generation*

It can be seen that the second part of the transition leading from state `S0` to state `S1` does not have any effect, because the opposing condition to the first one is used. This results in two failed objectives (25 and 31), which are listed in Table 3.34. Additionally, the transition from state `S1` to state `S2` is never executed due to the fact, that the same statement is already used with a lower execution order and therefore higher priority. Furthermore, the transition condition leading from state `S2` to `S1` is redundant and therefore also marked as non-achievable.

If one or more objectives cannot be reached, there must be a design error or wrong variable range. For the methodology and applications described in this thesis, automatically generated test cases have only been used to detect errors. For coverage analyses, manually written test cases for specific maneuvers and situations are used.

In the case of this state machine, those errors could have been found manually as well. However, as state machines get more and more complex the detection of errors gets increasingly difficult and resulting coverage analysis cannot be performed manually. Therefore, the test generation mode with coverage analysis is used extensively throughout all development steps of the methodology and applications described in this thesis.

Table 3.34: *Test Generation - Objectives (excerpt)*

#	Type	Description	Test Case
1	Decision	Chart: Substate executed State "S0"	1
2	Decision	Chart: Substate executed State "S1"	1
3	Decision	Chart: Substate executed State "S2"	1
4	Condition	Transition: Condition 1, "input_boolean == true" T	1
⋮	⋮	⋮	⋮
17	Decision	Transition: Transition trigger expression T	3
18	Decision	Transition: Transition trigger expression F	1
19	Decision	Transition: Transition trigger expression T	n/a
20	Decision	Transition: Transition trigger expression F	n/a
21	Decision	Transition: Transition trigger expression T	1
22	Condition	Transition: Condition 1, "input_boolean == true" T	2
23	Condition	Transition: Condition 1, "input_boolean == true" F	1
24	Condition	Transition: Condition 2, "input_uint8 >0" T	2
25	Condition	Transition: Condition 2, "input_uint8 >0" F	n/a
26	Decision	Transition: Transition trigger expression F	1
27	Decision	Transition: Transition trigger expression T	2
28	MCDC	Transition: Condition 1, "input_boolean == true" T	2
29	MCDC	Transition: Condition 2, "input_uint8 >0" T	2
30	MCDC	Transition: Condition 1, "input_boolean == true" F	1
31	MCDC	Transition: Condition 2, "input_uint8 >0" F	n/a

3.4.2.6 Property Proving

The SDV can also be used for property proving. In this mode, the inputs with specified ranges, are used to prove or disprove certain assertions. Those are normally implemented in a dissimilar way and optimally by another developer or tester.

Figure 3.36 shows a Stateflow chart in a Simulink environment with connected property proving assertions. In general, those are connected to the input as well as the output and include a dissimilar implemented function or part of the state machine. In this example, a slightly adapted version of the Edge Detector, which includes two minor errors (c.f. Subsubsection 3.3.2.6), is tested. The task of the Edge Detector is to detect a change of the input signal and reflect that in the output signal as explained in Subsection 3.1.3.

Therefore three assertions are used to guarantee its correct functionality in the subsystems `risingEdge`, `fallingEdge`, and `initialOutput`. They are located on the right side of Figure 3.36, colored in yellow, and consist of an alternative, in this case, *Simulink* implementation of one part of the state machine.

This *Simulink* example also includes buttons for running and opening the options of the SDV in the bottom part of the model in gray. They are used for convenience only during manual testing and are not necessary for the functionality itself.

However, due to the erroneous implementation of the Edge Detector in this example, the SDV can only verify one of the assertions and produces counterexamples for the other two. The modified state chart of the Edge Detector with two intentional errors is depicted in Figure 3.37.

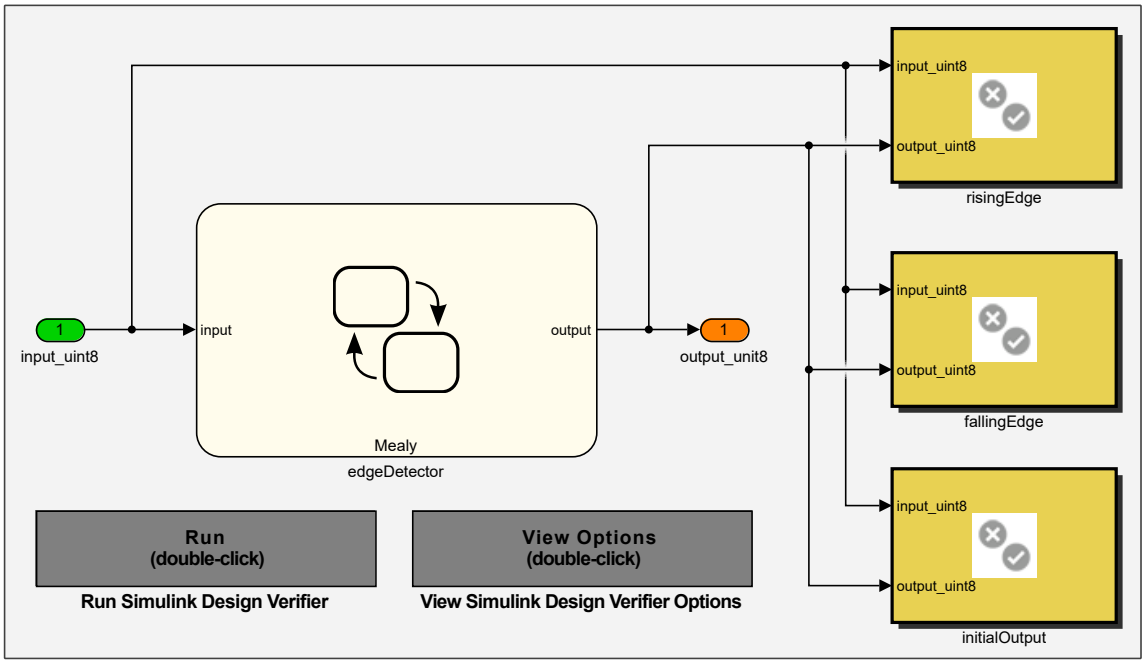


Figure 3.36: *Property Proving - Overview*

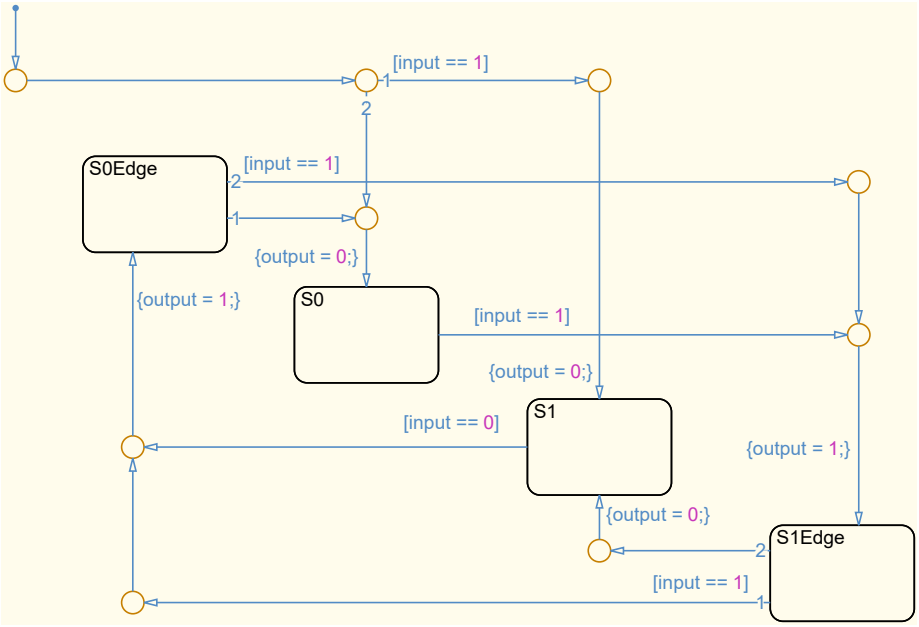


Figure 3.37: *Property Proving - Erroneous Edge Detector - State Machine*

The *Simulink* implementation for the assertion of the rising edge is depicted in Figure 3.38, where the rising edge is replicated in the reddish part. When the current input is "1" and the previous input, stored by the memory block, is "0", a rising edge has occurred.

However, due to the design, the model cannot detect those changes until the second time step, which is implemented in the greenish part with another memory block. Therefore, if a rising edge is detected in step two or later, this implies an output of "1". The combination of the "Implies" and the "Assertion" block is used to implement this.

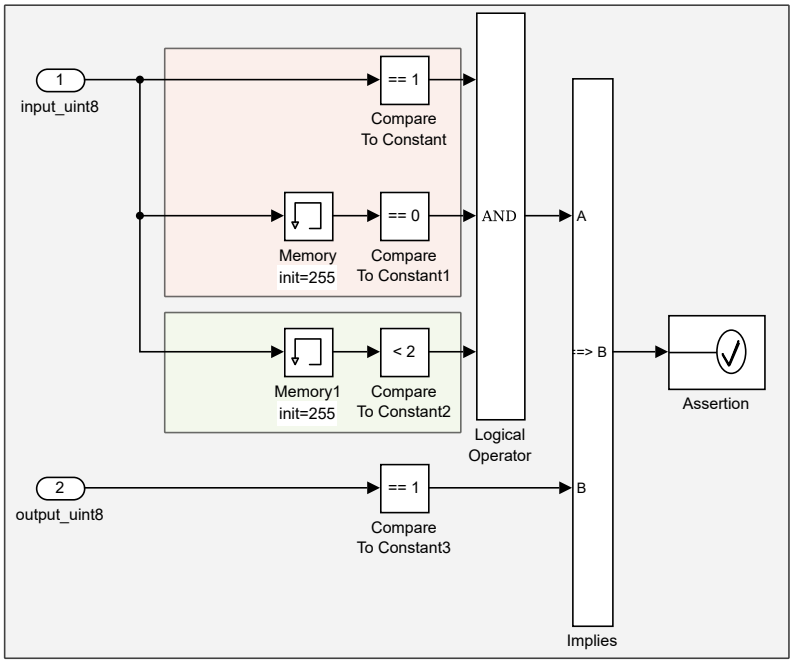


Figure 3.38: *Property Proving - Rising Edge Assertion*

An overview of the verified objectives is listed in Table 3.35. For the two disproved assertions, respective counterexamples are automatically generated and shown in Table 3.36 and Table 3.37 respectively.

The assertion for the falling edge is disproved with counterexample one. An input series of `0,1` in time steps one to two leads to an active `S1Edge` state (`init` → `S0` → `S1Edge`). Then the input changes to `0`, which due to a faulty transition condition from state `S1Edge` to `S0Edge`, leads to state `S1` with the output being "0". Therefore the falling edge from time step two to three has not resulted in the expected output of "1". Changing the transition from state `S1Edge` to `S0Edge` back to its original version (`[input==0]`) will resolve this problem (c.f. Subsubsection 3.3.2.6).

In a similar way the counterexample two for the rising edge assertion, with the input sequence `1,0` for time steps one to two leads to an active `S0Edge` state (`init` → `S1` → `S0Edge`). The next input is actually irrelevant because the transition from state `S0Edge` to `S0` has no condition and the highest priority execution order. Therefore, state `S0` will always be active one time step after `S0Edge`. Thus, the output will always be "0" even if a rising edge was used as input. Swapping the execution order of the two transitions leaving state `S0Edge` will resolve this problem as well and result in a correct implementation of the edge detector (c.f. Subsubsection 3.3.2.6).

Table 3.35: *Property Proving - Objectives*

#	Type	Model Item	Counterexample
1	Assert	fallingEdge/Assertion	1
2	Assert	initialOutput/Assertion	n/a
3	Assert	risingEdge/Assertion	2

Table 3.36: *Falling Edge Assertion - Counterexample 1*

Time	**0.00**	**0.01**	**0.02**
Step	**1**	**2**	**3**
input_uint8	0	1	0

Table 3.37: *Rising Edge Assertion - Counterexample 2*

Time	**0.00**	**0.01**	**0.02**
Step	**1**	**2**	**3**
input_uint8	1	0	1

3.4.3 Model in the Loop

Model in the Loop (MiL) simulations are used to connect the flight control modules for the first time. In this environment, their interaction with each other, as well as their closed-loop performance as FCC with an FDM, can be tested. The design, implementation, as well as testing, are all performed in *MATLAB*, *Simulink*, and *Stateflow*. Therefore, this framework serves as an integrated design environment and test harness at the same time. An overview of the test architecture, as used by all platforms, is presented in Figure 3.39.

The commands from the FCC, primarily actuator position commands, are sent to the FDM. The FDM is then calculating the new state of the aircraft and sending respective measurements back to the FCC. Control inputs for the FCC can be generated with a joystick, Mode Control Panel (MCP), or a test case. The generated data can be visualized, monitored, or logged and saved for later analysis.

The FCC includes flight control modules for the inner loop, autopilot, trajectory control, trajectory generation, automatic takeoff and landing, and system automation. Their functionality and especially the design, implementation, and testing of the system automation for experimental aircraft is described in Chapter 4.

The FDM includes an environment model, motion kinematics, the airframe of the specific aircraft, rigid-body equations of motion, and a model for aircraft systems. The environmental model, in turn, includes a ground, gravity, atmosphere (dynamic and static), and magnetic field model. The airframe part of the FDM includes models for actuation and flight mechanics, propulsion, aerodynamics, landing gear, and weight and balance. Additional systems like the actuator control electronics, the air data system, and the navigation system are modeled in the aircraft system part.

A more detailed example, of how MiL simulations are used as a cost- and time-efficient development and testing environment for the *DA 42* is available in [ZSMH2018].

3.4.4 Software in the Loop

Software in the Loop (SiL) simulations are a special case of MiL simulations. They use automatically generated C-Code versions of the flight control module or of the complete FCC. Therefore, the code generation process can be verified if compared to MiL results.

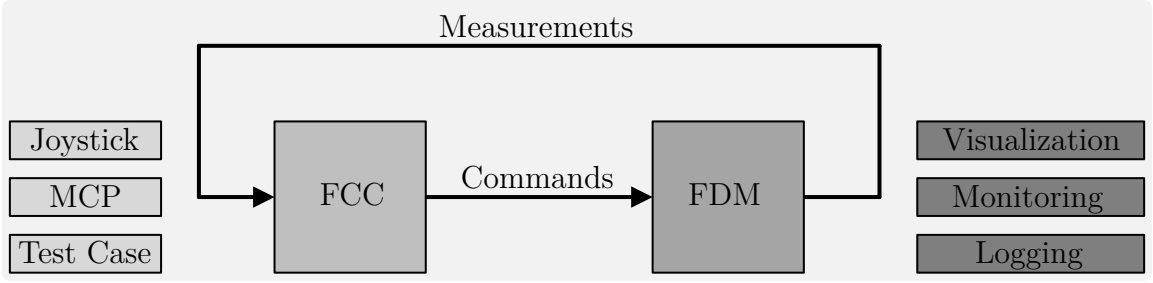


Figure 3.39: *Model in the Loop Verification*

3.4.5 Hardware in the Loop

The Hardware in the Loop (HiL) simulation is the first test that uses the real FCC with its flight control software. Before the test, the design models in *MATLAB*, *Simulink*, and *Stateflow* are used to generate C-Code. This is merged with an FCC-specific framework and transferred to the target hardware. The two main verification objectives are the FCC and its physical interfaces to adjacent aircraft systems as well as the code integration, compiling, linking, and flashing process. Tests comparing HiL to MiL or SiL results can be used to ensure the correct functionality of the interfaces and code-generation process.

In the HiL environment, the real FCC with genuine flight control software is used. Its outputs are connected via the physical interfaces to hardware that is capable of capturing and decoding those signals. The actuators and flight mechanics are modeled in software. The simulated deflections of the control surfaces are then fed into the FDM. The information about the new state of the aircraft from the FDM is then transferred back to the FCC via the physical interface and as being data from the navigation system.

Extensive HiL simulations have been conducted for *SAGITTA*, the *DA 42*, *ELIAS*, and the *Do 228* before proceeding with further tests. In the case of *SAGITTA*, HiL simulations were only considered passed, when the difference between HiL and MiL was small enough, which was used as one of the first flight clearance criteria [Kuc2018].

3.4.6 Aircraft in the Loop

In the next step, the complete aircraft is incorporated into the tests. The Aircraft in the Loop (AiL) simulations include not only the FCC but also actuators, flight mechanics, surfaces, engines, and communications. However, due to the fact that the aircraft is not moving, data concerning the movement, attitude, and position of the aircraft still needs to be simulated.

The most valuable experience from this AiL simulation is related to the actuators and the involved mechanical parts of the aircraft. Effects like offsets, biases, non-linearity, hysteresis, and noise of the involved sensors and components are hard to model, but the effects can be analyzed relatively easily using such simulations. Additionally, for optionally-piloted or manned aircraft, the flight crew can gain experience interacting with the Flight Control System (FCS).

Like the HiL simulations, the AiL tests were used with all four aircraft that are presented in Chapter 2 and are used as a demonstration platform for the applications presented in the following chapters of this thesis.

However, due to the missing movement, additional effects like dynamic pressure, wind, and vibration, and their effect on the surfaces are missing in such simulations. This is one of the reasons why the maneuver injection, described in Chapter 5, is developed. It can be used to perform highly accurate and repeatable flight tests to gain more information about the missing effects of the simulation.

3.4.7 Ground Tests

The Ground Tests are the first real-life tests with the complete aircraft, including motion sensors like the Inertial Navigation System (INS), data links, and a GCS. They are used to gain experience with the experimental aerial platform in a real-life environment but without the hazard of actual flight. They are particularly important for unmanned aircraft but can be used for manned and optionally-piloted aircraft as well.

In the case of the unmanned demonstrator *SAGITTA*, those ground tests were especially important because it had to perform its maiden flight fully automatically. Therefore, three ground test campaigns, in terms of taxi tests, were conducted, which included tests like single and double lane changes, straight acceleration runs, and back-up control law tests. However, the performance in terms of centerline tracking and automatic braking during the takeoff and touchdown phases could only be tested during the first flight. [SKHH2018, SH2017, SKH2017]

For the *DA 42*, a different ground roll centerline tracking controller was developed. It had to cope with the mechanical link between the rudder and nose wheel steering as well as the limited control authority of the latter at low speeds. [MZS+2017]

3.4.8 Flight Tests

The most challenging but also most exciting verification activities of aircraft are the Flight Tests. Especially when it's the first flight with the automatic flight control system, and even more so if the first flight has to be performed fully automatic from takeoff to landing.

Various aspects of the flight control systems and their adjacent components are tested and verified beforehand. However, certain effects cannot be simulated in the scope of experimental development projects and therefore some degree of uncertainty remains.

All experimental aircraft, described in Chapter 2 and used as demonstration platforms for the methodology and applications described in this thesis, performed their successful automatic first flight between 2016 and 2017. In the case of *SAGITTA* the system automation, described in Chapter 4 was used from the very beginning. Due to the optionally-piloted capabilities of the *DA 42*, the degree of automation was gradually increased. However, the system automation was part of the flight control software from the beginning as well, even if not with its full capabilities. With *ELIAS* and the *Do 228*, the maneuver injection was used throughout multiple flight campaigns to verify the FDM, analyze actuator performance and response characteristics, and for advanced development techniques like the model-based gain tuning of the controller.

Due to the special circumstances of *SAGITTA*, the planning, implementation, and execution of the fully automatic maiden flight were more complex, compared to the other demonstrators. Special precautions had to be taken regarding the real-time monitoring, ground station setup, and displays that were used to supervise the flight guidance and control system. [KSHH2018, KHH2018, KH2018]

3.5 Summary

This chapter presents the *Design, Implementation, and Testing Methodology for System Automation - using State Machines in Stateflow (MTSA).*

In the beginning, the theoretical basics of state machines and the background of the developed methodology are presented. This includes the history from the middle of the 20th-century to the modern descendants. Additionally, the theory of automata with Combinational Circuits, Sequential Machines, Finite State Automata, Transducers, Sequencers, Classifiers, Acceptors, Pushdown Automata, and Turing Machines is introduced. Furthermore, different state machine modeling techniques are presented including the Functional, Imperative, and Feedback System View as well as the Tabular, Graphical, and Matrix description. The first section of this chapter ends with an introduction of the most commonly used types of state machines, *Mealy* and *Moore*.

The second section deals with the design of state machines. It introduces the four common challenges of automation, being complexity, brittleness, opacity, and literalism. In the following, the eight-step design process of this methodology is explained. Hereinafter, the state machine internal and external decision logic, which is used for the arguments of the transition conditions, is presented. The section concludes with the design aspects of contribution *C1.1 - Hierarchical decomposition design strategy, minimizing complexity and optimizing testability.*

After previously presenting the design aspects, the next section is focusing on the implementation of the methodology. It introduces the toolchain, which uses a combination of *MATLAB*, *Simulink*, and *Stateflow*. Following this overview, the used elements in *Stateflow* are introduced, and an implementation example is given. After understanding the development environment the specific implementation of contribution *C1.1* is presented. The section is concluded with contribution *C1.2 - Modeling guidelines for implementation, minimizing opacity and maximizing software maintainability*, where all developed guidelines are presented.

The chapter ends with the presentation of testing and verification activities, which are also part of the developed methodology. After presenting Unit Tests, Model Checking and its application in this methodology is explained. In this part, the Simulink Design Verifier (SDV) and formal methods are introduced and their utilization for contribution *C1.3 - Incremental bottom-up application of formal methods, ensuring effective testing and guaranteed system characteristics* is presented. This includes the three operating modes of the SDV, being Design Error Detection, Test Generation, and Property Proving. In the following, other testing activities, which consist of Model in the Loop (MiL), Software in the Loop (SiL), Hardware in the Loop (HiL), and Aircraft in the Loop (AiL) are described. The description of Ground Tests and real-life Flight Tests concludes this section and the overall chapter.

Chapter 4

Flight Control System Automation

The developed *Operator-Centric Multi-User Flight Control System Automation for experimental UAVs and OPVs with Contingency Procedures (FCSA)* is presented in this chapter.

The FCSA is administering the cascaded control loop of the Flight Control Computer (FCC) and provides various operating modes. Those modes are normally controlled by the users, but can also be switched automatically, by the software, in case of degraded sensor performance or hardware malfunctions.

Besides the nominal functions, it features contingency procedures for both, operational as well as malfunction scenarios. It is important, that the switching between control modules or modes needs to be transient-free because otherwise the operator or pilot would be surprised by the system's behavior.

The FCSA was developed for Unmanned Aerial Vehicles (UAVs) and Optionally-Piloted Vehicles (OPVs). It is therefore tested on a UAV, the *SAGITTA* Research Demonstrator, and on an OPV, the *DA 42*. Platform type-specific modes, e.g. modes only for UAV or OPV, or project-specific modes are guarded so they cannot be activated on the wrong platform. Additionally, this requires that different use cases have to be represented within the same system and without re-implementation.

The methodology, described in Chapter 3, is applied during the design, implementation, and testing of the FCSA. During the design of the FCSA, the modes are separated into different levels. This modular approach makes it possible to enable or disable certain parts of the automation if not needed for a certain platform and makes future extensions relatively simple.

The modes are implemented using decomposed deterministic finite state machines utilizing *MATLAB*, *Simulink*, and *Stateflow*. The FCSA is tested using, among others, formal methods on software level as well as Aircraft in the Loop (AiL) simulations with the real aircraft. After passing the rigorous testing the real-life applicability of the FCSA is proven in flight tests with both demonstration platforms.

Motivation - Flight Control System Automation

- Higher-level flight control system automation is necessary to exploit the inherent benefits of UAVs and OPVs
- Currently available solutions are not suitable for experimental aircraft with multiple users
- No integrated contingency procedures are available in existing approaches, that support the operator in case of a failure

The current state of the art with respect to the system automation part of this thesis is introduced in Subsection 1.3.2. The resulting motivation, due to the identified deficits, is recapitulated here. Due to the variety of operational scenarios and training levels of the operators higher-level automation is necessary to exploit the benefits of UAVs and OPVs. However current solutions cannot cope with multiple users in the field of experimental aircraft. Additionally, current solutions, even for commercial airliners, mostly revert back to the human operator in case of failures. The deactivation of the automation increases the workload for the operator in times where the support by the automation is needed the most. This is not desirable for OPVs and might not even be possible for UAVs.

The derived objectives, from the deficits in the current state of the art, are presented in Subsection 1.4.2 and summarized in the following. The overall objective is the automatic administration of flight control loops of experimental UAVs and OPVs. Due to incidents and accidents caused by *Loss of State Awareness*, which is also referred to as *Mode Confusion*, human-centered automation has been considered for a long time. However, to this day the problem still exists and is even more problematic for physically separated operators and pilots. Due to the development context of experimental aircraft, which includes the integration of multiple users with access to various control levels, the problem is increased even more. Therefore, additional procedures for robust automation shall be developed, that support the operator and pilots in case of malfunctions.

Objectives - Flight Control System Automation

- Implement a system automation to administer a cascaded flight control loop of experimental UAVs and OPVs
- Provide a operator-centered automation with interfaces for multiple users, like operators and pilots
- Develop procedures for robust automation, even in non-nominal operational circumstances, targeting operator workload reduction

Contributions - Flight Control System Automation

- C2.1 - Strategy for switchability between various modes on different authority levels, enabling experimental automation
- C2.2 - Operational management concept for multi-user experimental OPVs and UAVs, increasing mode awareness
- C2.3 - Automatic operational and malfunction contingency procedures for continuous operation in non-nominal circumstances

The contributions of the system automation, presented in this chapter, are introduced in Subsection 1.5.2. The assignment of the individual contributions to the sections of this chapter is presented in the following together with the general outline.

At the beginning of this chapter, Section 4.1 describes the hardware and *System Architecture* used for the two demonstration platforms. The FCSA is designed for and tested on the UAV *SAGITTA* and the OPV *DA 42*. Part of the contribution *C2.1* is the integration of the FCSA within the FCC architecture, as described in this section.

The *Operation Modes* and *Transition Conditions and Actions* are presented in Section 4.2 and Section 4.3. This includes a description of the purpose for each mode as well as their transition conditions and actions to understand the interactions with surrounding elements and other modes. Those two aspects constitute contribution *C2.2*.

Various contingency procedures are integrated into the FCSA. Those consist of operational, concerning mission changes, as well as malfunction, for sensor failures, procedures to guarantee continuous automatic operation. The *Loiter Automation (LA)* is described in Section 4.4 as an example of contribution *C2.3*.

The FCSA needs to access all hierarchy levels within the cascaded control loop, from directly controlling the surface actuators all the way to activating separate modules for automatic takeoff and landing. Therefore entry points between each adjacent control loop need to be defined. Those points are command switches that are controlled by the FCSA and for each entry point, outputs from higher-level control loops, dynamic values generated by the FCSA, or static values can be used. The second part of contribution *C2.1* is presented in Section 4.5. It builds on contribution *C1.1 - Hierarchical decomposition design strategy, minimizing complexity and optimizing testability* and describes those *Injection Switches* and their integration within the cascaded flight controller.

Real-life *Flight Tests* are presented in Section 4.6, which proves the applicability of the developed software before the chapter is concluded with a *Summary*, in Section 4.7, of the archived contributions and resulting possibilities of the FCSA.

The basic idea for this FCSA and parts of it have been previously published [KH2016, KH2017b, KH2018b]. However, this chapter takes a more detailed approach and includes various undisclosed aspects of the system automation.

4.1 System Architecture

This section introduces the platform-specific hardware architectures as well as the generic software architecture of the *Operator-Centric Multi-User Flight Control System Automation for experimental UAVs and OPVs with Contingency Procedures (FCSA).*

One main requirement is the platform-independent design and transferable implementation, without changing the internal structure, so as not to affect prior testing and verification. Therefore the FCSA is tested on one $150kg$ Unmanned Aerial Vehicle (UAV) and one four-person twin-engine Optionally-Piloted Vehicle (OPV), as described in Chapter 2. The sensor overview of the latter is depicted in Figure 4.1. The hardware architecture, including an introduction of all sensors, is presented in Subsection 4.1.1.

In the following, the generic Flight Control Computer (FCC) system architecture and all flight control modules are introduced (Subsection 4.1.2). Those are used for automatic takeoff and landing, generating and controlling the trajectory, basic autopilot and autothrottle functions, and low-level control. Additionally, modules for input conditioning and monitoring as well as output processing are included. They are all controlled by the FCSA, which constitutes part of contribution *C2.1 - Strategy for switchability between various modes on different authority levels, enabling experimental automation.*

In Subsection 4.1.3 the software architecture of the FCSA module is presented as an example of how all flight control software modules are encapsulated and connected. This is a crucial design aspect for the usage in various platforms and configurations.

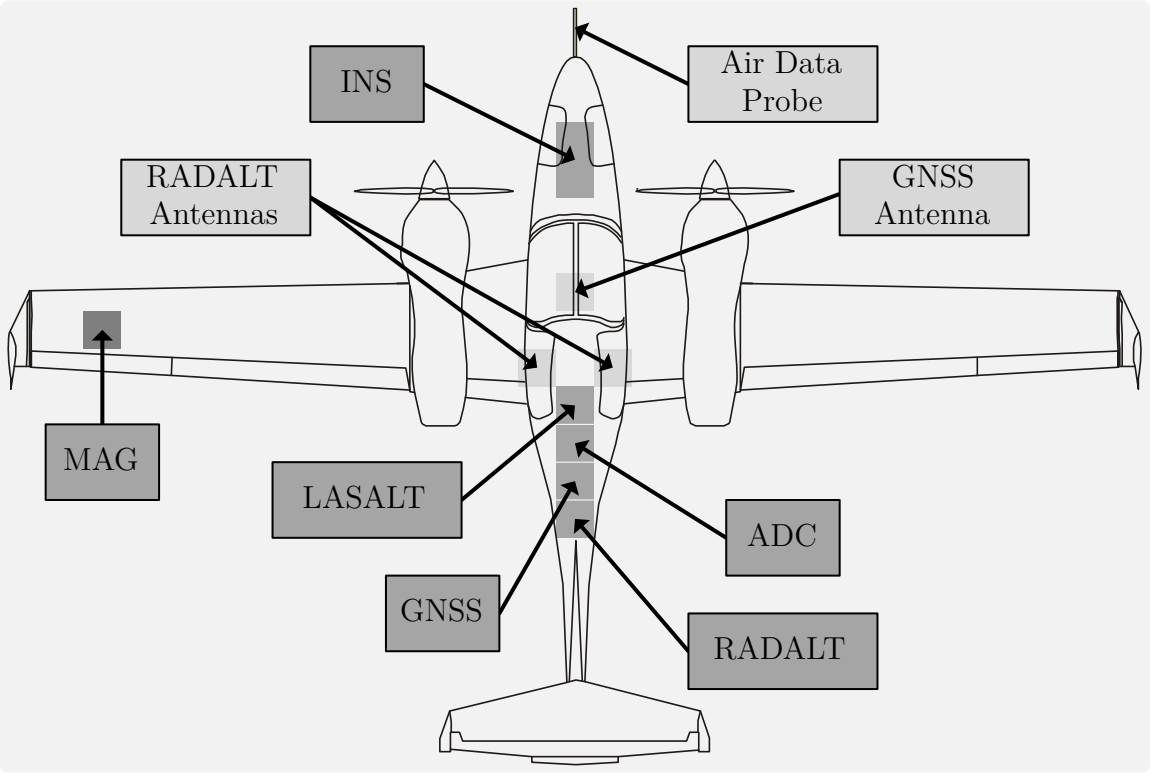


Figure 4.1: *DA 42 Sensor Overview (contour from [Dia2012, p. 38])*

4.1.1 Hardware Architecture

The hardware architecture, in this part, refers to the FCC and the hardware connected to it. These can be sensors to get information about the aircraft's state, data links to receive data from and sent data to the Ground Control Station (GCS), as well as actuators to control the surfaces of the aircraft, and many more.

Depending on the aircraft and project, different configurations and combinations are used. The FCSA is designed for UAVs and OPVs. Consequently, it is tested on both types, which are introduced in Chapter 2. On the one hand, *SAGITTA* is a $150kg$ UAV demonstrator with a wingspan of $3m$ and two integrated jet turbines.

On the other hand, the *DA 42* is a modified four-seater Part 23 Class II aircraft with a wingspan of about $14m$ and a maximum take-off mass (MTOM) of $1999kg$. Despite their completely different specifications, a fairly similar, with respect to FCC connections, hardware architecture is used. The project-specific hardware architectures of *SAGITTA* and the *DA 42* are presented on the following pages.

Both platforms use a similar implementation of a cyclic execution sequence, within the FCC to gather information and transmit commands. As an example, an overview of the different tasks in the FCC of *SAGITTA* is depicted in Figure 4.2. It uses a Periodic Interval Timer (PIT) to time each calculation cycle, which in all demonstration platforms referenced in this thesis is $10ms$, which results in an update rate of $100Hz$.

At the beginning of each cycle the data, received during the last cycle, is requested from the Input/Output Controller (IOC). It is packed and sent to the main Central Processing Unit (CPU). There the data is extracted, stored in input registers, the flight controller algorithm is executed, the data is copied from the output registers, and packed and transmitted back to the IOC. The hardware is then monitored, while the data is being transmitted. All those steps need to be completed well before the $10ms$ have elapsed in order for the next cycle to start as planned. [NHH2017, p. 2ff]

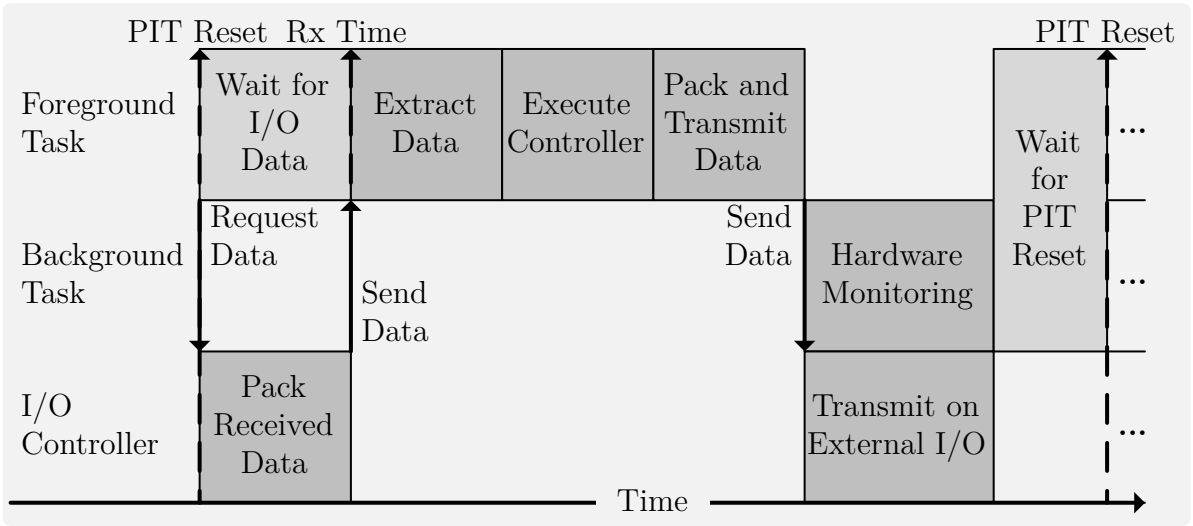


Figure 4.2: *FCC Task Overview (adapted from [NHH2017, p. 2])*

4.1.1.1 SAGITTA Architecture

SAGITTA, introduced in Section 2.1, is the UAV on which the FCSA is demonstrated. As a basis for discussion in this chapter, in the following the hardware architecture is explained in more detail.

A simplified hardware architecture with the FCC and its connections is depicted in Figure 4.3. Inputs from sensors are shown on the left, available data links above, and mainly outgoing connections on the right of the FCC.

The main source for aircraft measurements is the Inertial Navigation System (INS), with integrated Global Navigation Satellite System (GNSS) and (external) Magnetometer (MAG). They provide information about, among others, the aircraft's turn rates, attitude, and position. Additionally, an Air Data Computer (ADC) is used to gather data like airspeed and barometric altitude. Due to the lack of a commercial-grade landing system like an Instrument Landing System (ILS), an additional Radar Altimeter (RADALT) is needed for ground referencing during automatic takeoff and landing.

The aircraft is equipped with three data links. The main Command and Control (C2) data link for the Flight Operator (FO) is the Flight Data Link (FDL), the Mission Data Link (MDL) is mostly used for mission data but also includes C2 data as a backup, and the unidirectional External Pilot Data Link (EPDL) is used by the External Pilot (EP).

The aircraft is controlled using eight trailing-edge-flaps. Those are connected to actuators, which in turn are controlled by eight Actuator Control Electronics (ACEs). The two inboard flaps are used as the elevator, the two mid-board flaps as the aileron, and the two pairs of outboard split-flaps as the rudder. The two turbines are controlled via the Engine Control Units (ECUs). The nose and two main landing gears are controlled by the Gear Control Electronics (GCEs), which include commands for the brakes as well.

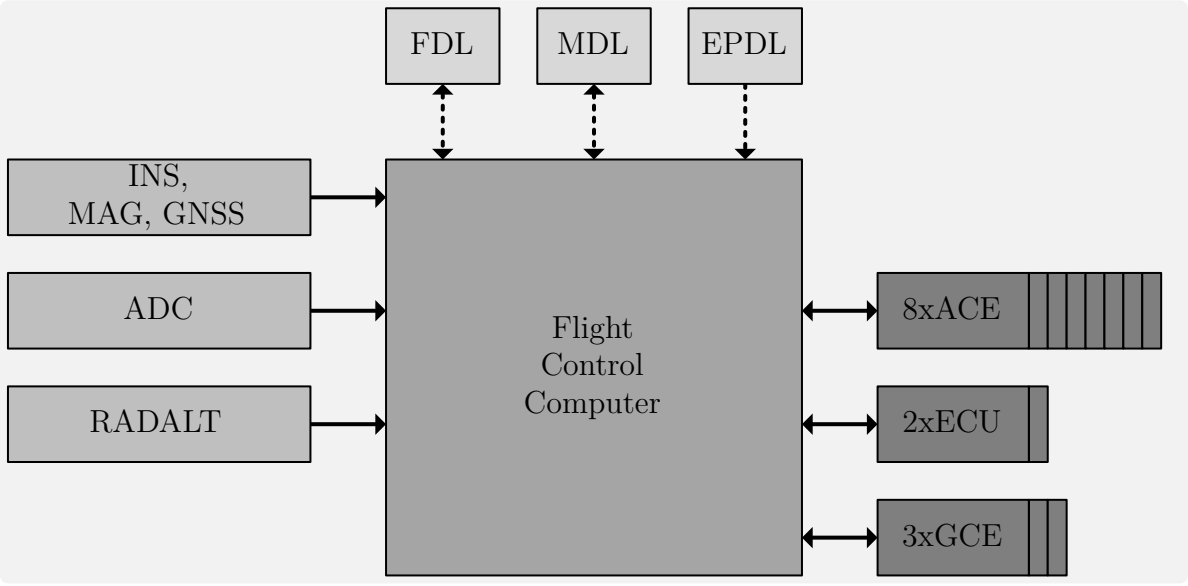


Figure 4.3: *SAGITTA Hardware Architecture*

4.1.1.2 DA 42 Architecture

The *DA 42*, introduced in Section 2.2, is the OPV on which the FCSA is demonstrated. As a basis for discussion in this chapter, in the following the hardware architecture is explained in more detail.

A simplified hardware architecture with the FCC and its connections is depicted in Figure 4.4. Inputs from sensors are shown on the left, available data links above, and mainly outgoing connections on the right of the FCC.

Information about the movement, attitude, and position of the aircraft is sent to the FCC by the INS and its internal MAG and GNSS. Airspeed and barometric altitude are measured by the ADC. Additionally, a RADALT and Laser Altimeter (LASALT) are used to create a ground reference. Complementary data, like control surface positions, are collected and measured by the Data Concentrator Unit (DCU).

The Shared Data Link (SDL) is used as a C2 data link between the GCS and the aircraft. It transfers command data to the aircraft and returns monitoring data to the ground for supervision. Additionally, data can not only be displayed on a Multi-Function Display (MFD), but it can be also used as additional command input, as a replacement for, or in addition to the data link.

Computed positions for the control surfaces are converted and sent to the actuators. The *DA 42* uses four actuators for the main control surfaces and two for the throttle levers of the engines. The Electromagnetic Clutches (EMCs) of the four main actuators for the elevator, aileron, rudder, and elevator trim are guarded and monitored by the Control and Monitoring System (CMS). All actuators return position data to the FCC and CMS for monitoring purposes.

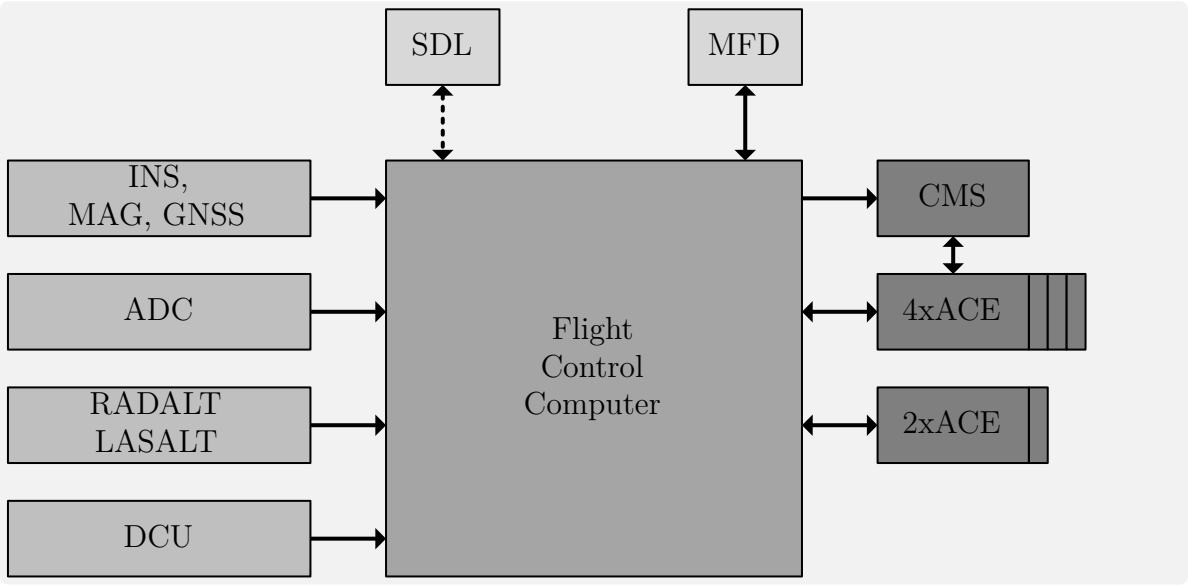


Figure 4.4: *DA 42 Hardware Architecture*

4.1.2 FCC System Architecture

Following the previously introduced hardware architecture, in this part, the software architecture within the FCC and all flight control modules are introduced. For *SAGITTA* and the *DA 42*, slightly different software architectures are used, which is addressed in the respective description of the modules themselves. An overview of all modules and the command injection of the FCSA is depicted in Figure 4.5. The switching modules (TG-, TA-, IL-, and AC-SW), in part, constitute contribution *C2.1 - Strategy for switchability between various modes on different authority levels, enabling experimental automation.*

The FCC consists of up to six flight control modules and two additional processing modules. They are listed here and explained in more detail on the following pages. From the *System Automation (SA)* to the *Inner Loop (IL)* those control modules get more aircraft-specific and less generic.

- *Input Processing and Monitoring (IPM)*
  Preprocessing of input signals; Compiling of system-specific buses; Monitoring of input signals based on reasonable ranges, changes over time, or validity indications
- *System Automation (SA)*
  Administration of cascaded control loops; Interface provision for multiple users; Control of injection switches to perform various maneuvers
- *Automatic Takeoff and Landing (ATOL)*
  Utilization of lower-level control loops to perform automatic takeoff and landing; Contingency procedures for close ground proximity scenarios
- *Trajectory Generation (TG)*
  Generation of three-dimensional trajectories; Generates distances and deviations from flight plans for lower-level control loops
- *Trajectory Control (TC)*
  Aircraft control based on high-level control like distances generated by TG; Generates angular and load-factor commands for Inner Loop
- *Auto Flight Control System (AFCS)*
  Aircraft control based on medium-level controls like altitude, heading, and speed; Generates angular and load-factor commands; Includes *Autothrottle (ATHR)*
- *Inner Loop (IL)*
  Basic control and stabilization of the aircraft; Includes control surface allocation; Generates angular deflections for the control surfaces
- *Output Processing (OP)*
  Computing of additional data and performing interface scheduling; Generation of system-specific and physical interface dependent output buses

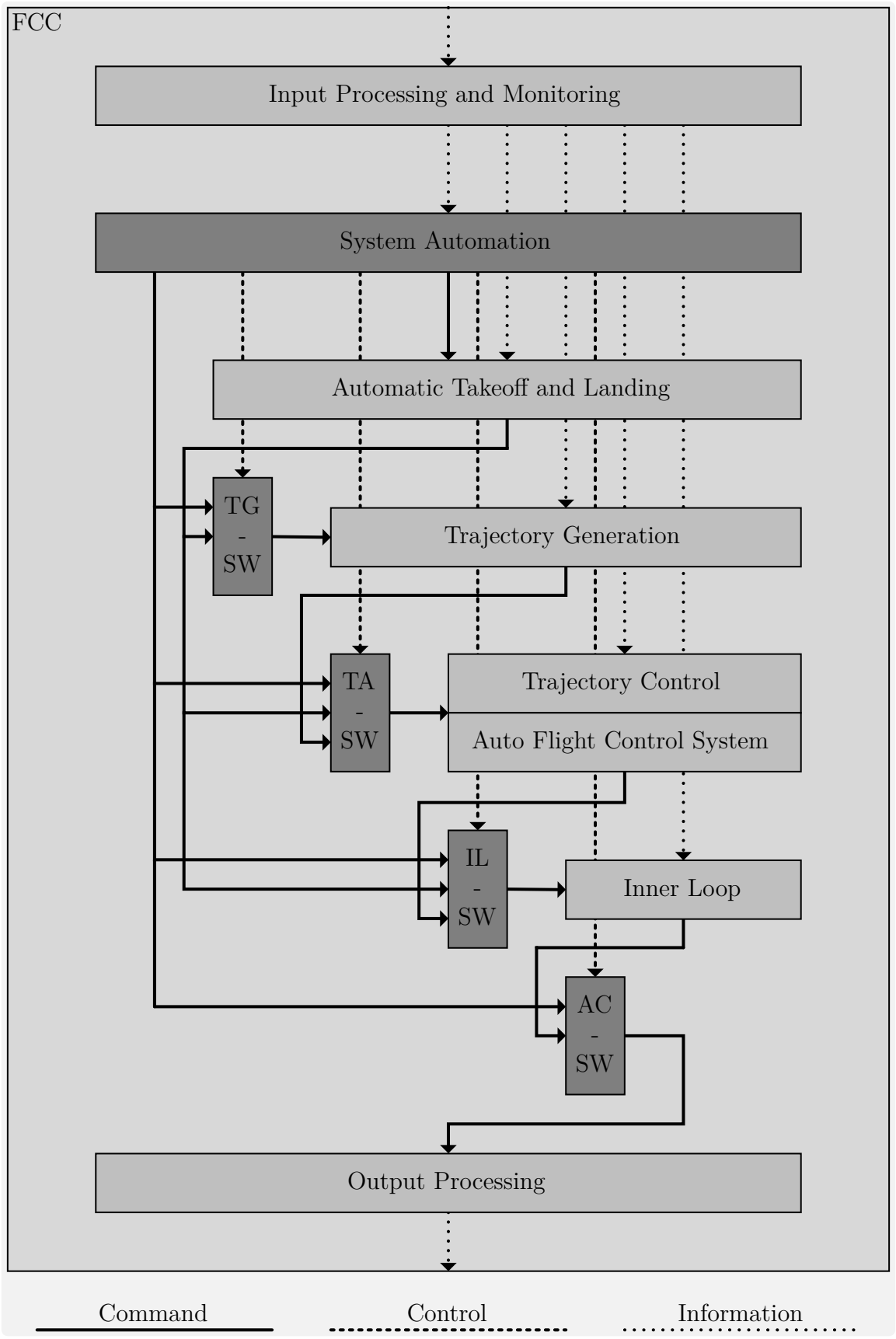


Figure 4.5: *FCC System Architecture*

4.1.2.1 Input Processing and Monitoring

The *Input Processing and Monitoring (IPM)* is not a flight control module, but the first part of the flight control algorithm of the FCC, where all data, that is received from the physical interfaces is processed. This data includes commands from FO in the GCS or the EP, sensor data from the INS, MAG, ADC, or others, and information from other aircraft systems like control surface positions.

Objectives of the IPM include integrity evaluation, sensor filtering, signal selection for redundant information, as well as compiling of system-specific buses. The integrity is evaluated by Cyclic Redundancy Checks (CRCs), signal ranges, and analyzing the data over time. Depending on the signal, the data is only forwarded to the other modules if the tests are passed or validity indications are added to the signal buses.

Depending on the type and source of the signal, some sensor data is also filtered or altered in other ways. High-frequency noise is removed from low-quality signals, while coordinate system transformations are applied to others to provide coherent data. Additionally, an "age counter" is added to various signals. Many signals are not updated every $10ms$, the cycle time of the FCC, therefore a counter is added that indicates the elapsed cycles since the last update.

In the case of *SAGITTA*, where multiple data links are used, the IPM is also selecting the available or most recent available source, for redundantly available data. All data is then grouped and transformed from physical interface-driven sensor definitions to flight control-driven generic buses. Additionally, a monitoring, fault detection, and diagnosis module is used to provide integrity information [KH2015]. Those are then forwarded to the six flight control modules, which are introduced in the following.

4.1.2.2 System Automation

The *System Automation (SA)*, is administering other flight control modules, with respect to commands from the users, based on sensor availability or data integrity. In the following, SA, instead of FCSA, will be used to denote the specific flight control module. Figure 4.5 shows a simplified version (feedback and advanced flow information is omitted) of the interaction between the SA and the other flight control modules of the FCC. Commands are indicated by solid lines, switch controls by dashed lines, and other information by dotted lines. In part, this constitutes contribution *C2.1 - Strategy for switchability between various modes on different authority levels, enabling experimental automation.*

The SA is controlling the command flow throughout the FCC using injection switches. The *Trajectory Generation - Switch (TG-SW)*, *Trajectory Control / Auto Flight Control System - Switch (TA-SW)*, *Inner Loop - Switch (IL-SW)*, and *Actuator - Switch (AC-SW)* are used to change the inputs and redirect the output of flight control modules. Since the SA has a command and a control input to those switches, the alternative input can also be generated by the SA itself. Those switches are explained in more detail in Section 4.5.

4.1.2.3 Automatic Takeoff and Landing

The *Automatic Takeoff and Landing (ATOL)* module is performing the first and last part of an automatic mission. It is responsible for controlling the aircraft during the transition from ground to air and vice versa, as well as while in close ground proximity. The module is split into two parts handling the takeoff and landing separately.

The ATOL module of *SAGITTA* and its parts for takeoff and landing are based on a phase-breakdown of both maneuvers. They use a state machine-based approach, which in turn utilizes the existing guidance and control modules to perform fully automatic takeoff and landing. The module and its phases include various aspects of the methodology presented in Chapter 3 of this thesis. For all connecting phases, that is the beginning and end of both the takeoff and landing, there are respective modes within the SA, which allow for continuous automatic operation. [KH2016b, KH2016a, KH2017b, KH2017c, KH2017a]

The ATOL module of the *DA 42* and their takeoff and landing implementation is also based on a phase breakdown. However, the phases, transitions, and utilized flight control loops differ slightly. Due to the fact that flight test data was available from previously conducted manual takeoffs and landings the controller could be designed close to the behavior of the pilot. Additionally, based on various manual flight tests, decrab phases are integrated into the landing controller to allow for non-ideal landing situations. [MH2017, MKHS2017, MSH2017, ZMW+2017]

The ATOL module is the only flight control module, besides the SA, which has a connection to all other modules. Due to this special case, there is no injection switch for ATOL but a close collaboration of both modules during takeoff and landing to cover a variety of cases. This is described in the respective parts of Section 4.2 and Section 4.3.

4.1.2.4 Trajectory Generation

The *Trajectory Generation (TG)* module is an online three-dimensional trajectory generation system. It is based on the Global Positioning System (GPS) and has two main tasks. On the one hand, it needs to determine a flight path that is attainable by the aircraft and at the same time fulfills constraints like desired altitude, course angle, and speed. On the other hand, it needs to compute the error dynamics between the aircraft and the trajectory using relative kinematics. The calculations are based on flight plans, which consist of various waypoints. Besides their GPS position, those points include additional parameters like altitude and speed, different ARINC 424 leg types [AEC2011] which are oriented on a subset of RTCA DO-236C [RTC2014], different transitions between those like fly-by and fly-over, end of flight plan indications, as well as loiter definitions. [MSH2015, SMH2015, SPS+2016, SH2017, GSL+2018]

The TG switch is used to switch between three basic types of inputs. Those consist of commands from the SA, commands from the ATOL module, and default inputs for deactivating the TG. The switch is described in more detail in Subsection 4.5.1.

4.1.2.5 Trajectory Control

The *Trajectory Control (TC)* module is tightly connected to the previously introduced TG. The controller is derived based on the nonlinear dynamic inversion of second-order nonlinear error dynamics between a specified trajectory and the path of the aircraft.

The TC module is designed to drive the deviations in the lateral and vertical plane to zero and therefore can be used by multiple higher-level modules like TG and ATOL.

It uses various inputs from its adjacent module, which include the deviations between the aircraft and the desired trajectory, the corresponding time derivatives of these deviations, desired trajectory angles, angular rates, angular accelerations, and desired kinematic acceleration at the trajectory reference point.

The TC and also the AFCS module provide normalized specific force commands in the kinematic frame with respect to unaccelerated flight, i.e. curvature commands in each axis, to the next lower module.

Even though the controller is able to control the kinematic velocity as well, the function is not utilized because the airspeed is controlled by the next loop. [SGGH2018, SH2017, SSK$^+$2016, SH2014]

4.1.2.6 Auto Flight Control System

The *Auto Flight Control System (AFCS)* has basically two operational modes. On the one hand, it can utilize the force commands from the TG and on the other hand, it uses the desired altitude, heading, and speed commands to provide basic autopilot functions. When the aircraft is following a three-dimensional trajectory the AFCS calculates a body-fixed load-factor and roll-angle command from the specific-force commands of the TG. When in "autopilot mode" the AFCS performs a dynamic inversion of the flight path dynamics and uses reference models to compute the commands for the IL. Commands are separated into vertical, lateral, and energy. Vertical commands include pitch-angle, vertical-speed, flight-path-angle, and altitude, while lateral commands include roll-angle, heading, and track.

Additionally, an *Autothrottle (ATHR)* is incorporated, which is used to control the airspeed. The ATHR loop uses kinematic measurements and an engine model to control the airspeed. However, the "autopilot mode" as well as the ATHR control loop require GPS to work properly. For increased robustness, a backup law is also implemented that drops this requirement, albeit with reduced performance. The ATHR module is able to control different speeds like Indicated Air Speed (IAS) commands from the FO, speeds that are set via the waypoints, specific throttle percentages, or even the speed with respect to the ground, i.e. ground speed. [KSH$^+$2017, KHB$^+$2017, KGSH2016, KSB$^+$2016]

The TA switch is used to control the input of both, the TC and AFCS simultaneously. It has five different switch positions, which are described in more detail in Subsection 4.5.2.

4.1.2.7 Inner Loop

The *Inner Loop (IL)* is the most aircraft-specific flight control module because it includes the control allocation and delivers direct commands to the actuators.

In the case of *SAGITTA*, the IL is a Control and Stability Augmentation System (CSAS) because the aircraft requires stability-enhancing feedback. It consists of three parts which are the longitudinal and lateral controller as well as the control allocation module. Depending on the control modes it follows commands from the control loops or from the EP. The longitudinal controller has two different command inputs, a load-factor command, and a pitch-angle command. The outer flight control loops utilize the load-factor command while the EP can directly command the pitch-angle. The lateral controller provides a bank-angle command, which is used in all modes. The assignment of the elevator, aileron, and rudder commands to the eight control surfaces of *SAGITTA* is performed within the control allocation. [BWB+2012, GH2012, BGHH2014]

In the *DA 42*, due to the inherent stability of the aircraft, a different IL is used to mainly archive consistent command following over the entire flight envelope. It uses the same load-factor and roll-angle command but includes an additional lateral load-factor as input. The latter is normally set to zero but can be utilized in situations like cross-wind landings, where a decrab maneuver is necessary. Additionally, the internal feedback structure of the *DA 42* IL is completely different from the one in *SAGITTA*. [GHSM2021]

The IL is highly dependable on the aircraft's configuration. This is founded in the aircraft dynamics as well as surface architectures and corresponding allocation. However, this also means that all other loops are not at all or much less dependent on the aircraft. This increases the general validity of the overall concept and is one of the strengths of this modular and cascaded flight control loop approach.

The injection switch for the IL, which is controlled by the SA, utilizes eight different sets of commands which are explained in Subsection 4.5.3.

4.1.2.8 Output Processing

The *Output Processing (OP)* module is used to generate the hardware-specific interface buses for the ACEs, ECUs, and others. Additionally, it calculates complementary data like CRCs, which is not required within the cascaded control loop, but necessary before sending data on the physical interface. Furthermore, it can be used for scheduling or splitting messages over different time slots. That is necessary if the physical interface is not capable of transferring all data within one cycle. This lowers the data rate, e.g. from $100Hz$ to $50Hz$, but allows for twice as many variables to be transmitted.

Its inputs are controlled by the SA via the AC-SW. It can be used to prevent the flight control loops from accessing the actuators in special cases, which are explained in Subsection 4.5.4.

4.1.3 Software Module Architecture

After previously introducing the FCC system architecture and all flight control modules, in the following, an explanation of the software architecture of the module itself is given. An overview of how each flight control module is integrated within the FCC software is depicted in Figure 4.6.

They all consist of an *Input Handling* and *Output Handling* part as well as the *Core Module* itself. This architecture is used to enable the use of modules in various projects and system architectures. Each project has aircraft-specific sensors, as introduced in Subsection 4.1.1, and therefore also aircraft-specific buses with signals that might not exist in other projects.

Therefore, project-specific buses are translated into module-specific buses in the *Input Handling* block. This part of the software module needs to be designed and adapted for each system architecture. Additionally, to translating from aircraft to module-specific buses, all signals that are irrelevant for the specific module are not copied, thus reducing the number of variables within each module. On the other side, the module-specific buses are translated back to the aircraft-specific buses. This enables generic modules, that can be used in various projects, without adapting the module itself.

The SA and other modules as well, use enumerations for various modes and variables within the module itself. As described in Subsection 3.3.4, they are used to pass state information through all levels of the multi-level state machine. However, due to compatibility issues, they are not used outside of the respective software module. To be able to pass the information from one module to another, they are therefore converted from integers to enumerations in the *Input Handling* and from enumerations to integers in the *Output Handling*.

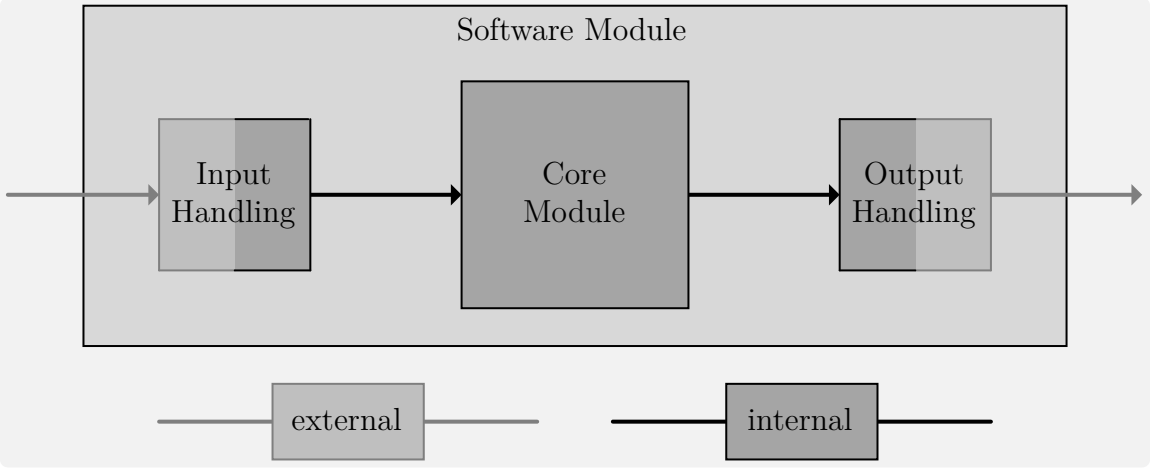


Figure 4.6: *Software Module Architecture*

4.2 Operation Modes

This section describes the different modes that are available within the SA. This includes the operation mode itself, their respective level, and possible superposition options. Figure 4.7 depicts a hierarchical overview of them. In total, this includes 15 operation modes, distributed across four levels and four superposition options. This constitutes the first part of contribution *C2.2 - Operational management concept for multi-user experimental OPVs and UAVs, increasing mode awareness.*

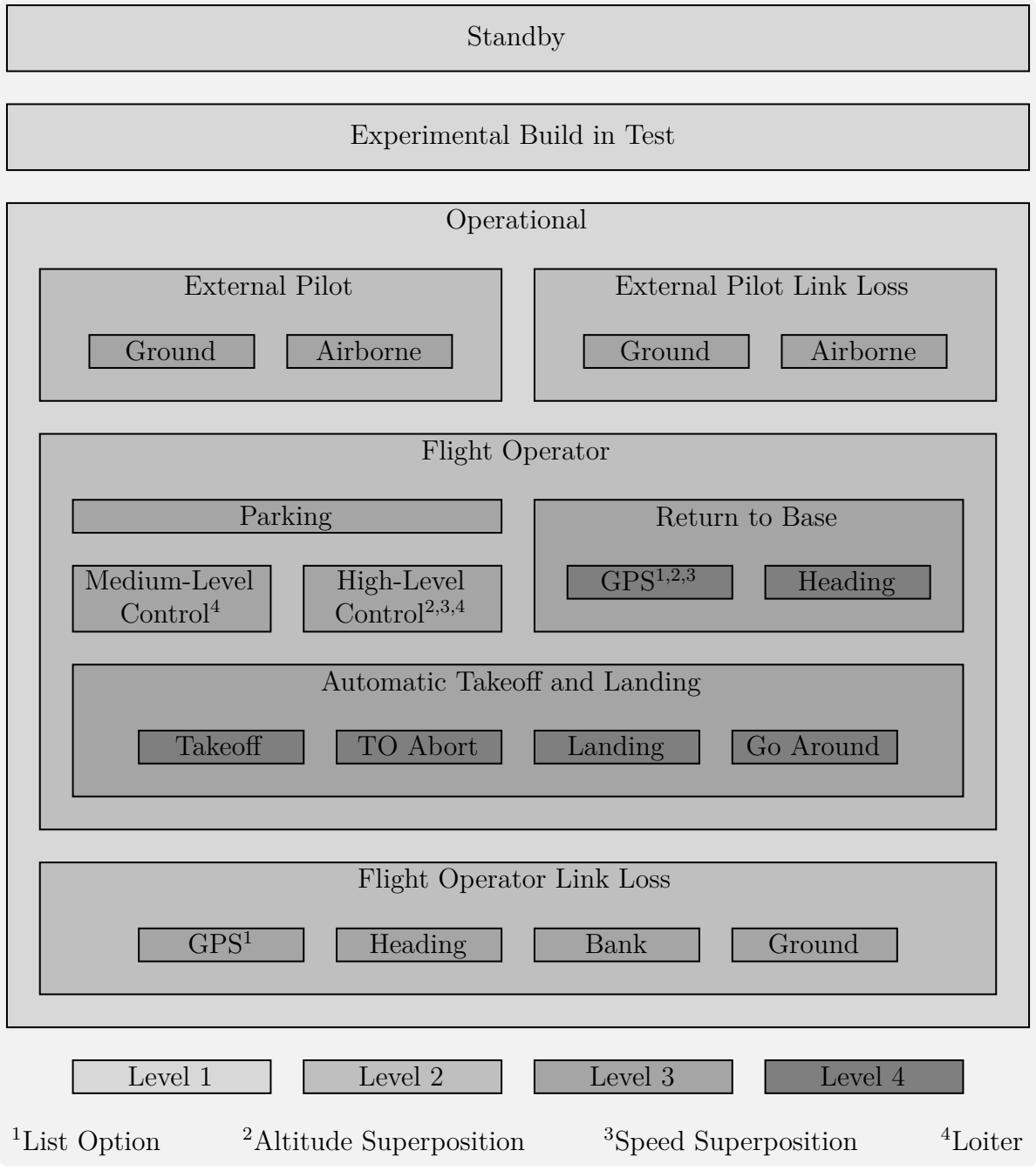


Figure 4.7: *Mode Level Overview*

4.2.1 Level 1

Level 1 of the SA includes three basic modes of operation: *Standby* (*STB*), *Experimental Build-in-Test* (*XBIT*), and *Operational* (*OPL*).

4.2.1.1 Standby

Standby (*STB*) is used, whenever the aircraft is not controlled automatically. Depending on the use case, this can happen in different scenarios.

In the case of a UAV, this can only happen during the preparation of the aircraft on the ground. This basic statement leads to a high effort for design and implementation. It has to be assured that the mode cannot be activated, while the UAV is airborne or during takeoff and landing on the ground, as this would lead to the loss of the aircraft.

When using the SA in an OPV, *STB* can be active when being on the ground as well as when being airborne. While in the air, it covers the time when the pilot is controlling the aircraft. In this case, the Flight Control System (FCS) is powered, but not controlling the aircraft and the pilot is using the classical mechanical control system, with the clutches being open.

4.2.1.2 Experimental Build-in-Test

The *Experimental Build-in-Test* (*XBIT*) is used for integration tests, hardware tests, and other non-operational tests. Direct-law refers to a control method, where the stick deflection of the pilot is directly mapped to the deflection command of the surface. For the UAV demonstrator *SAGITTA*, a control in direct-law is not possible, due to the shape of the aircraft. Nevertheless, a direct-law is implemented, which can be used in *XBIT*. This allows for pre-flight tests of the complete transmission from pilot input and sending the data to the aircraft over calculating surface positions within the FCC and sending them to the actuators to the correct operation of the actuators and the movement of the respective control surfaces.

Due to the experimental nature of this test mode, it has to be assured, just like with *STB*, that it is not inadvertently activated during normal operation.

4.2.1.3 Operational

The default mode on *Level 1* is *Operational* (*OPL*). It is the parent mode for all other modes when the FCS is active. When used in a UAV environment this mode is automatically activated after power on, if no other command, for either *STB* or *XBIT*, is present. This is done to mitigate the effects of in-flight power resets and to return to normal operation as soon as possible. In an OPV use case, such effects are not that critical. The following subsections describe the child modes of *OPL*.

Handling of the aircraft on the ground, which requires either *STB* or *XBIT*, is not influenced since both modes can be activated via the respective command.

4.2.2 Level 2

Level 2 of the SA deals with the distinction between the two basic types of operators, EP and FO, and their respective malfunction strategies for link loss. The four modes on *Level 2* are introduced in the following.

In either case, the contingency mode is automatically activated if a link loss is detected. However, the normal mode is not automatically reactivated if the link is restored. Such an reactivation has to be triggered by the respective pilot.

4.2.2.1 Operational - EP

OPL_External-Pilot (*OPL_EP*) is the normal operation mode for the EP, which is active when the EP is controlling the aircraft and no link loss is detected. The EP can take over control of the aircraft by using a switch on the remote control or by pressing a button on a joystick, depending on the use case.

4.2.2.2 Operational - EP Link Loss

OPL_External-Pilot Link Loss (*OPL_EPLL*) is automatically activated by the SA if a link loss is detected while the EP is controlling the aircraft. Depending on the configuration this decision is based on different data links and information about their integrity and availability. In the case of *SAGITTA*, the EP has a separate data link, therefore *OPL_EPLL* is activated if this link fails. In the case of the *DA 42*, the EP can either be onboard the aircraft, thus not using a data link, or in the GCS using the SDL. Depending on the circumstances this mode is activated based on the integrity of the data link or inhibited if an onboard EP is controlling the aircraft. Furthermore, it can also be suppressed, by overwriting data link integrity information, if the onboard pilot takes over in case of an EP link loss.

4.2.2.3 Operational - FO

Similar to *OPL_EP*, *OPL_Flight-Operator* (*OPL_FO*) is the normal operating mode for the FO, which is active when the FO is controlling the aircraft and no link loss is detected. The FO can take over control, by pressing a button in the GCS. Whoever, EP or FO, presses the button last is in control of the aircraft. In the rare event of a simultaneous activation, the EP is given priority.

4.2.2.4 Operational - FO Link Loss

Just like *OPL_EPLL*, *OPL_Flight-Operator Link Loss* (*OPL_FOLL*) is automatically activated if the FO is controlling the aircraft and a link loss is detected. In the case of *SAGITTA*, this decision is based on two data links, FDL and MDL. Therefore, this mode is only entered if both data links fail. Within the *DA 42*, it is only based on the SDL.

4.2.3 Level 3

Level 3 of the SA is used throughout all *Level 2* modes. On this level, the modes are mainly divided based on the flight phase or operational objective. Due to the high number of modes on this level (not in one state machine), they are introduced based on their *Level 2* parent mode.

4.2.3.1 Operational - EP

The *Level 2* mode *OPL_EP* contains two *Level 3* modes, *OPL_EP_Ground* (*OPL_EP_GND*) and *OPL_EP_Air* (*OPL_EP_AIR*). Depending on the airborne status of the aircraft, one or the other is automatically selected. This state can be calculated in different ways. The SA has an internal calculation, based on Weight on Wheel (WoW) sensors. It can also use an external signal, which can be calculated by another control module within the FCC or by an external system like a navigation system.

Ground The *OPL_EP_GND* mode is designed for the EP to control the aircraft during takeoff or landing roll. In the case of *SAGITTA*, a specific direct-law-like control mode with rate damping is used. When used in an OPV like the *DA 42* different control laws are used depending on the location of the EP. If the EP is onboard the aircraft a direct-law is used for control, while a higher-level control law is used when the EP is in the GCS to account for the higher latency introduced by the data link.

Airborne *OPL_EP_AIR* is used when the EP is controlling the aircraft while airborne. Similar to the *OPL_EP_GND* mode, different control strategies are used depending on the platform. While *SAGITTA* uses a rate-command-attitude-hold control law with separated thrust command, the *DA 42* uses direct-law or rate-command-attitude-hold when the EP is onboard the aircraft and a curvature-law when the EP is in the GCS.

4.2.3.2 Operational - EP Link Loss

Just like *OPL_EP*, *OPL_EPLL* contains two child modes, *OPL_EPLL_Ground* (*OPL_EPLL_GND*) and *OPL_EPLL_Air* (*OPL_EPLL_AIR*), which depend on the same airborne status of the aircraft.

Ground If a link loss occurs during takeoff or landing, while the aircraft is still on the ground, *OPL_EPLL_GND* is automatically activated. In this mode the control surfaces are centered, thrust is set to idle, and brakes are applied.

Airborne *OPL_EPLL_AIR* will be automatically activated if a link loss takes place in the air while the EP is controlling the aircraft. In this case, the SA will store the current heading and use it as a command alongside a predefined (safe) altitude and speed.

4.2.3.3 Operational - FO

The *Level 2* mode *OPL_FO* contains five *Level 3* modes, *OPL_FO_Parking* (*OPL_FO_PARK*), *OPL_FO_Medium-Level-Control* (*OPL_FO_MLC*), *OPL_FO_High-Level-Control* (*OPL_FO_HLC*), *OPL_FO_Return-to-Base* (*OPL_FO_RTB*), and *OPL_FO_Automatic Takeoff and Landing* (*OPL_FO_ATOL*). Those can be selected by the FO but are checked for availability and can be automatically reverted in case of certain malfunctions.

Another mode, which is used by the automation but not visible to the operator is *OPL_FO_ATOL-Unconfirmed* (*OPL_FO_ATUC*). It is explained in Subsubsection 4.3.3.2.

Parking *OPL_FO_PARK* is used either before takeoff or after landing. While preparing for takeoff or waiting for clearance this mode can be used to wait. Similarly, it can be used after landing, to shut down the aircraft from a defined mode. *OPL_FO_PARK* is similar to *OPL_EPLL_GND* because the surfaces are set to zero deflection, the thrust is set to idle and the brakes are applied.

Medium-Level-Control If the FO wants to control the aircraft while airborne with respect to autopilot commands, *OPL_FO_MLC* can be activated. A commonly used set of commands for all axes is altitude, heading, and speed. Other sets are also possible, which are described in the relevant sections.

High-Level-Control The mode *OPL_FO_HLC* provides GPS-based waypoint flight. It can be selected by the FO if GPS is available. Consequently, it will automatically be disabled if a GPS loss occurs. In this case, *OPL_FO_MLC* will be activated again. During *OPL_FO_HLC* the GCS is required to use the currently measured altitude, heading, and speed of the aircraft and forward them as commands. Therefore the aircraft will continue in a straight and level flight if an automatic downgrade to *OPL_FO_MLC* is necessary, with the benefit of synchronized commands.

Return-to-Base One of the operational contingency modes is *OPL_FO_RTB*. It can be used for a quick return of the aircraft to the base without manually changing the heading or selecting an appropriate waypoint list. The automatically determined child modes are described in Subsubsection 4.2.4.1.

Automatic Takeoff and Landing If takeoff or landing should be executed by the FCS, *OPL_FO_ATOL* must be active on *Level 3*. The aircraft will then perform an automatic takeoff or landing, based on the selected child mode. Those are explained in Subsubsection 4.2.4.2.

4.2.3.4 Operational - FO Link Loss

The *Level 2* mode *OPL_FOLL* contains four contingency *Level 3* modes, *OPL_FOLL_GPS* (*OPL_FOLL_GPS*), *OPL_FOLL_Heading* (*OPL_FOLL_HDG*), *OPL_FOLL_Bank Angle* (*OPL_FOLL_BANK*), and *OPL_FOLL_Ground* (*OPL_FOLL_GND*).

GPS If a link loss happens while the FO is controlling the aircraft, the SA automatically switches to *OPL_FOLL_GPS*. In this mode, the aircraft flies along a predefined GPS-based link loss waypoint list, which leads it to its home base.

This is done from all modes regardless of control level, e.g. *OPL_FO_MLC* or *OPL_FO_HLC*, due to the better performance in this mode.

Heading If a GPS loss occurs, while the aircraft is in *OPL_FOLL_GPS*, it automatically switches to *OPL_FOLL_HDG*. Since waypoint-based flight is no longer possible, a heading towards to home base is used. While in *OPL_FOLL_GPS*, the SA continuously calculates the heading based on the current position. When switching to *OPL_FOLL_HDG*, the last calculated heading is used, alongside predefined values for altitude and speed. Without disturbances, this will lead the aircraft to the home base as well and in the event of interference, it at least results in reducing the distance and thus increasing the probability of reestablishing a connection to the aircraft.

Resolving the GPS loss does not lead to a re-activation of *OPL_FOLL_GPS*, due to the fact that the used waypoint lists can only be started at the first point. Therefore, this could lead to an endless switching between those two modes which in turn would not bring the aircraft any closer to the home base. Nevertheless, if the GPS loss can be resolved during *OPL_FOLL_HDG*, the heading to the home base is updated, which mitigates the effects of disturbances and is described in more detail in Subsubsection 4.2.4.1.

Bank If a link loss occurs after a GPS loss, *OPL_FOLL_BANK* is automatically activated, because *OPL_FOLL_HDG* cannot be used since the position of the aircraft is unknown. In this mode, a predefined altitude, bank-angle, and speed are used to force the aircraft into a slow upward spiral, thus increasing the possibility of the link being reestablished. Eventually, this leads to a circular pattern at the defined altitude as an upper limit. If the GPS loss is resolved an automatic transition to *OPL_FOLL_GPS* is executed.

Ground If a link loss occurs, while the FO is in command and the aircraft is on the ground, *OPL_FOLL_GND* is automatically activated. The only possible situation, where this can happen, is during *OPL_FO_PARK*. Nevertheless, it is necessary to have this mode, because there is no other suitable *OPL_FOLL* child mode. Similar to *OPL_EPLL_GND*, the control surfaces are centered, the thrust is set to idle and the brakes are applied.

4.2.4 Level 4

The only modes that are directly controlled by the SA on *Level 4* are child modes of *OPL_FO_RTB*. The other modes are part of ATOL but are also described here to give a better overview of the functionality and close interaction of the SA and ATOL.

4.2.4.1 Operational - FO - RTB

OPL_FO_RTB is one of the operational contingency modes, which can be used by the FO to return the aircraft to its home base, without manual intervention, thus reducing workload in abnormal situations. It has two *Level 4* child modes: *OPL_FO_RTB_GPS* (*OPL_FO_RTB_GPS*) and *OPL_FO_RTB_Heading* (*OPL_FO_RTB_HDG*).

GPS *OPL_FO_RTB_GPS* is activated if GPS is available when the FO activates *OPL_FO_RTB*. This mode is similar to *OPL_FOLL_GPS* and also uses a GPS-based waypoint list to guide the aircraft to its home base.

If GPS is not available *OPL_FO_RTB* is rejected by the SA because neither waypoints can be used nor can a valid heading be calculated. This can only be the case if the automation is in *OPL_FO_MLC* and in this case, it will stay in this mode.

Heading A GPS loss is tolerated during *OPL_FO_RTB_GPS*, resulting in an automatic transition to *OPL_FO_RTB_HDG*. Just like *OPL_FOLL_HDG*, it uses a calculated heading towards the home base and a predefined altitude and speed. Consequently, a re-activation of *OPL_FO_RTB_GPS* is also not possible due to the same reasons discussed in Subsubsection 4.2.3.4. However, here as well as in the other contingency modes, the heading is updated if GPS becomes available again.

In Figure 4.8 three different scenarios, with the influence of wind, are shown. Pointing the aircraft's nose towards the destination is also referred to as homing in aerial navigation.

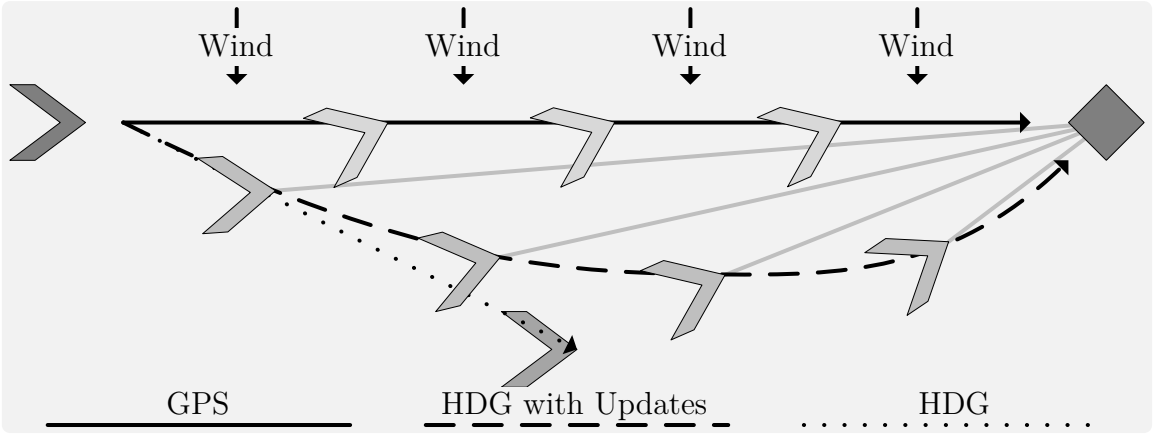


Figure 4.8: *Homing*

4.2.4.2 Operational - FO - ATOL

Due to the inherent special environmental conditions, the functionality for automatic takeoff and landing is integrated into a separate flight control module. It has its own state machine and different modes, which also utilize the underlying control modules. [KH2016a, KH2017a]

The SA and ATOL module are separate systems but are tightly interconnected. Due to this fact, the SA mirrors the modes to provide a uniform interface to the operator. The modes are self-explanatory and therefore only briefly introduced here to provide the necessary background to understand the reaction from the SA.

Takeoff *OPL_FO_ATOL_Takeoff* (*OPL_FO_ATOL_TO*) can be activated by the FO if the aircraft is on the ground, on the runway, within certain heading limits, and inside a few other restrictions. The aircraft then performs a fully automatic takeoff, without any necessary intervention from the pilot up to a defined waypoint.

Takeoff-Abort If a malfunction is detected during *OPL_FO_ATOL_TO* the respective contingency mode, *OPL_FO_ATOL_Takeoff-Abort* (*OPL_FO_ATOL_TOABRT*) is automatically activated. This mode can also be triggered directly by the FO, in case of an event that cannot be detected by the automatic system. In this mode, the aircraft is stopped as fast as possible on the runway.

If the aircraft has exceeded the takeoff decision speed or is already airborne, the activation of *OPL_FO_ATOL_TO* is inhibited. In this case, the FO has to use alternative modes to return to the airport as soon as possible.

Landing Towards the end of a mission, *OPL_FO_ATOL_Landing* (*OPL_FO_ATOL_LAND*) can be activated by the FO, if the aircraft is in a certain corridor and flying towards the runway. The alignment with the runway, capturing of the glide slope towards the ground, and landing itself are then performed fully automatically.

Go-Around *OPL_FO_ATOL_Go-Around* (*OPL_FO_ATOL_GOARND*) is automatically triggered by the system or can be alternatively activated by the FO in the GCS. In this mode, the landing attempt is aborted and a climb towards a safety altitude is initialized.

During *OPL_FO_ATOL* the automatic link loss procedure of the SA is inhibited. This makes the automatic activation of *OPL_FO_ATOL_TO* and *OPL_FO_ATOL_GOARND* possible in the first place because otherwise, the SA would overwrite those settings as soon as a link loss occurs. At the end of either of those *OPL_FO_ATOL* specific contingency modes, the SA and its general contingency maneuvers automatically take back control of the aircraft.

4.2.5 Additional and Superposition Options

Depending on the currently active mode, special options can be used, which are explained in the following and marked in superscripts in Figure 4.7. They are part of *OPL_FO_MLC*, *OPL_FO_HLC*, *OPL_FO_RTB_GPS* and *OPL_FOLL_GPS*.

4.2.5.1 List Option

Depending on the shape of the operational area it can be beneficial to use more than one list for a quick return of the aircraft. Therefore, the *List Option* can be used in *OPL_FO_RTB_GPS* and *OPL_FOLL_GPS*. During normal operation, the FO can choose between two different lists as required by the area and/or position of the aircraft. In case one of the modes gets activated the currently chosen list is used to guide the aircraft to its home base.

An example, where different lists are useful is shown in Figure 4.9. It depicts the runway to the left of a keep-out area, like a city, and the mission's objective to the right. In this case, one list starts in the middle of the left area, while the other starts on the right side. By switching to the correct list, depending on the current position of the aircraft, it can be assured, that the keep-out area is not violated in case of a link loss.

4.2.5.2 Altitude Superposition

When *OPL_FO_HLC* or *OPL_FO_RTB_GPS* is active, *Altitude Superposition* can be enabled. In this case, the vertical guidance is switched from waypoint-based to altitude-based. Therefore, the altitude command, as used in *OPL_FO_MLC*, is applied for vertical control of the aircraft.

4.2.5.3 Speed Superposition

Similar to *Altitude Superposition*, *Speed Superposition* is also possible in *OPL_FO_HLC* or *OPL_FO_RTB_GPS*. Both options can be used separately or together, the latter leaving waypoint-based guidance only active for the lateral motion of the aircraft. In this case, the speed command, as used in *OPL_FO_MLC*, is applied to the ATHR.

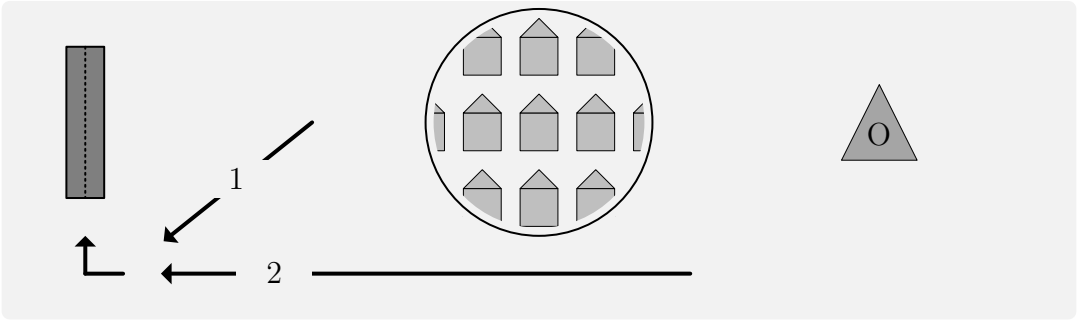


Figure 4.9: *Mission Area*

4.2.5.4 Loiter

If the FO wants to "pause" the mission, a holding pattern can be used in *OPL_FO_MLC* and *OPL_FO_HLC* at any time during the mission, by activating *Loiter*. An overview of the six modes within the *Loiter Automation (LA)* and their allocation to the *Level 3* modes is depicted in Figure 4.10.

Since those two modes use different control levels, autopilot commands for the former and GPS-based waypoints for the latter, depending on the active mode, different child modes of *Loiter* are activated.

Loiter_Bank (*LTR_BANK*) is activated when requiring a holding pattern within *OPL_FO_MLC*. In this case, the loiter pattern is based on a predefined bank-angle. Other command values, e.g. altitude and speed, are not modified and can be adjusted by the FO. An exit from this mode is executed immediately.

On the other hand, *Loiter_GPS* (*LTR_GPS*) is activated, when currently in *OPL_FO_HLC*. This mode uses a GPS-based holding pattern. Since both, *Altitude Superposition* and *Speed Superposition* are available in *OPL_FO_HLC*, they are also available in addition to *LTR_GPS*. The entry and exit to this mode are done in a deterministic way using *Loiter_Unconfirmed* (*LTR_UNCO*), *Loiter_Will Exit* (*LTR_WIEX*), and *Loiter_End-of-Flight-Plan* (*LTR_EoFP*).

This functionality was developed as an encapsulated automation within the SA. It reflects part of contribution *C2.3 - Automatic operational and malfunction contingency procedures for continuous operation in non-nominal circumstances* and is described in more detail in Section 4.4.

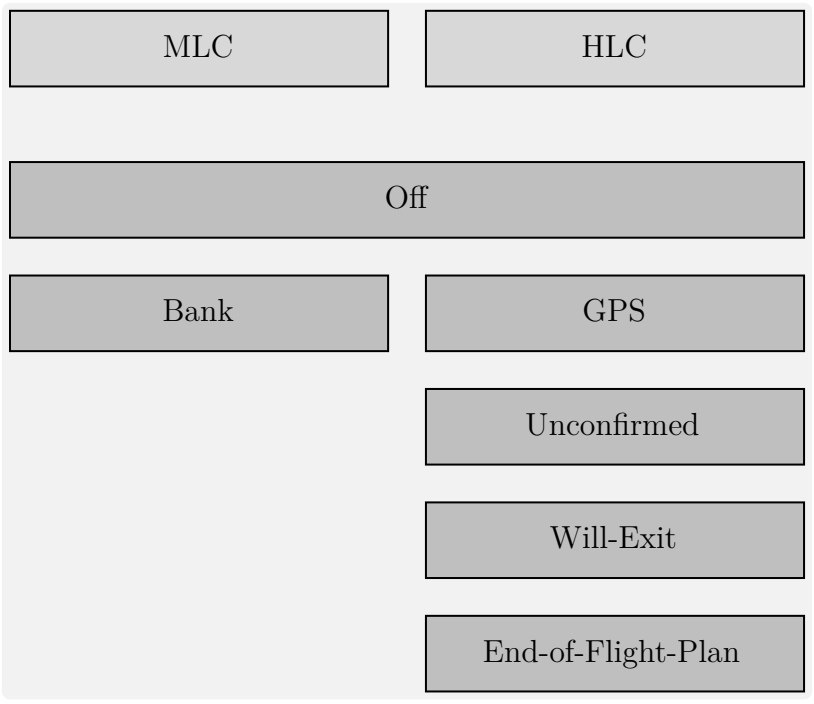


Figure 4.10: *Loiter Modes - Overview*

4.3 Transition Conditions and Actions

This section describes the transition conditions and actions between all operational modes on the different levels as introduced in Section 4.2. It is associated with the second part of the contribution *C2.2 - Operational management concept for multi-user experimental OPVs and UAVs, increasing mode awareness.*

All parts of this section include the state machine of the respective level, the interface table, and the transition matrix. In the state machine, the conditions and actions are encapsulated within *Stateflow* boxes and therefore not visible. They are explicitly listed in Appendix D. The interface tables present an overview of all input and output variables used by the state machine. This includes their name, direction, datatype, and range. The transition matrix shows which transitions in the state machine are possible.

A disassembled level view, where each line represents one state machine with its respective modes is depicted in Figure 4.11.

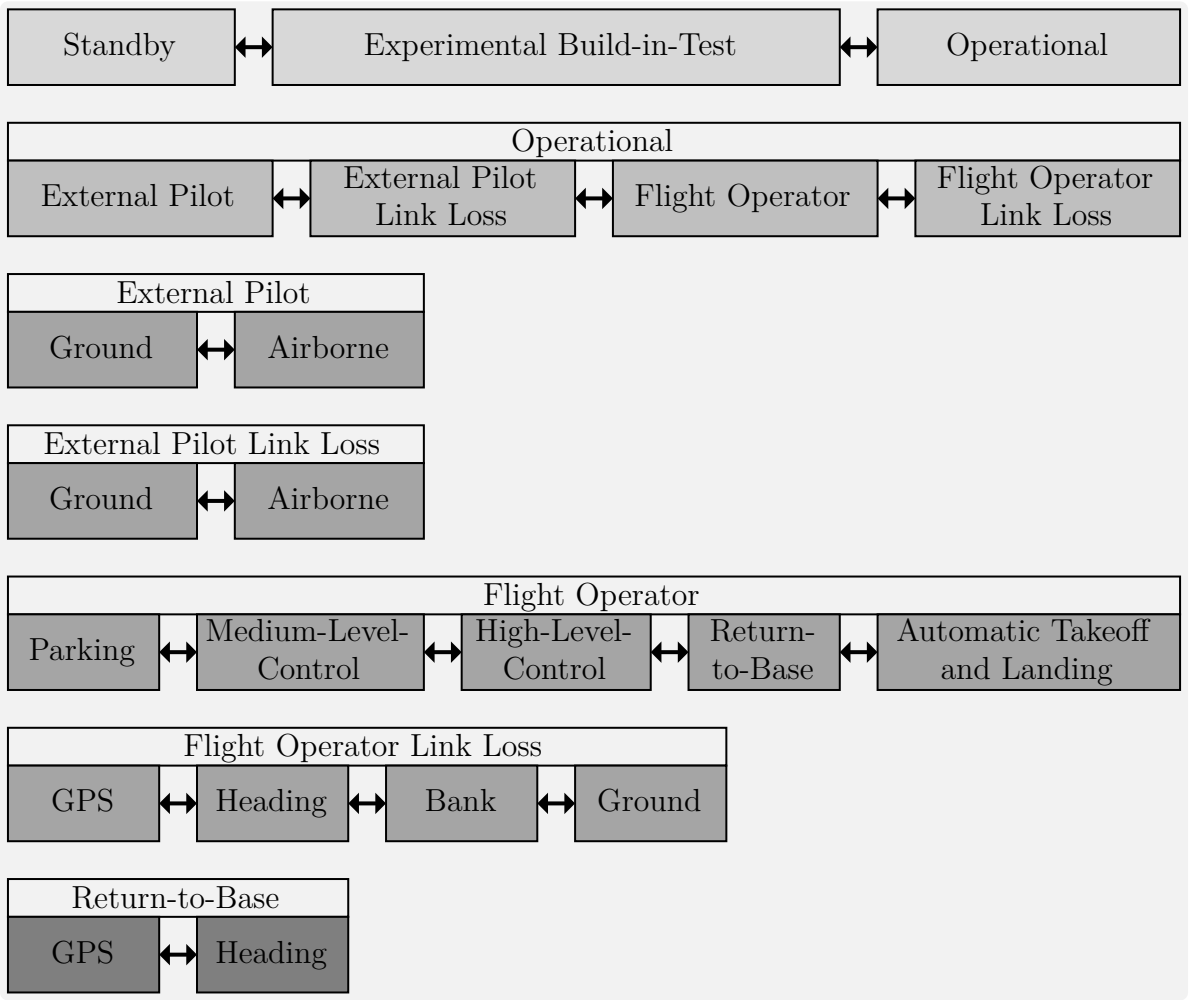


Figure 4.11: *Mode Level Overview - Disassembled*

Since a direct assignment from parent to child modes is not possible in this representation (c.f. Figure 4.7), the respective parent mode is shown above each line for better comprehension of the respective operation modes.

The transition conditions and actions between those modes on each level are described in this section. Due to the decomposition, enabled by the generic design, implementation, and testing methodology described in Chapter 3, significantly fewer conditions and actions have to be defined. This is also evident by the fact, that there are seven state machines, but with fewer (a maximum of five) modes.

In the following, all transitions are denoted in a "from→to" notation. If a transition is used multiple times, with multiple "from" or "to" states that part is denoted as "x".

4.3.1 Level 1

The *Level 1* of the SA includes *Standby* (*STB*), *Experimental Build-in-Test* (*XBIT*), and *Operational* (*OPL*), which are the three basic operational modes of the aircraft. As explained in Subsection 4.2.1, *OPL* is used for normal operation, *XBIT* for special procedures, and *STB* in all other cases.

The *Stateflow* state machine is depicted in Figure 4.12. It shows the three modes and possible transitions. The used input and output variables are listed in Table 4.1. Additionally, the possible transition and their respective names are shown in Table 4.2.

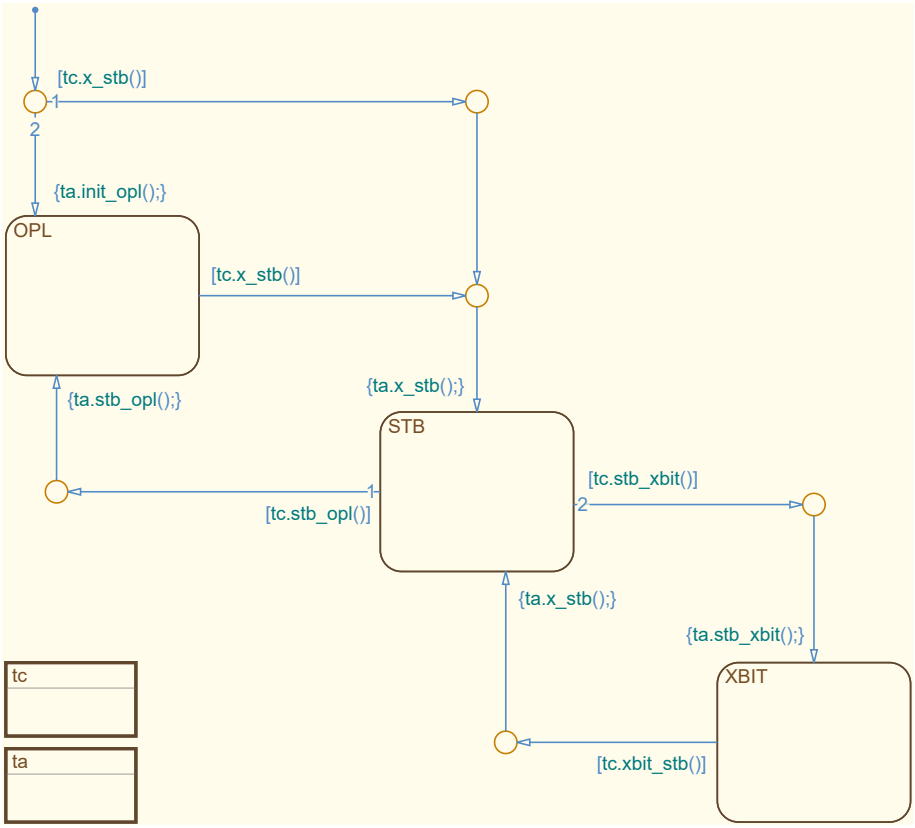


Figure 4.12: *Level 1 - State Machine*

Table 4.1: *Level 1 - Interface*

Name	**Direction**	**Datatype**	**Range**
`stb_flg`	input	boolean	0, 1
`xbit_flg`	input	boolean	0, 1
`level1_lgx`	output	enum	0, 1, 2, 3
`recovery_flg`	output	boolean	0, 1

At initialization, two transitions, *init→stb* and *init→opl* are possible. The transition to *STB* is taken if `stb_flg` is `true`. Otherwise, *OPL* will be active. This is a very critical transition, which is most relevant to UAV operation. During the startup procedure, a device is used, which is able to send a specific message that only includes this signal. Therefore, it is ensured, that it is never received in flight. This is important for being robust against an in-flight power reset, where the FCC automatically needs to get to its operational state. Since this does not reflect a normal startup procedure, the `recovery_flg` is set to `true` if the transition to *OPL* is taken. Additionally, the `level1_lgx` is set to its respective values of one for *STB* and three for *OPL*, depending on the active state.

Assuming *STB* is active, the normal startup procedure would continue with *stb→opl*. This transition is activated if `stb_flg` is set to `false`. In this case, *OPL* is activated, the `level1_lgx` is set accordingly and the `recovery_flg` is set to `false`. The latter is done to revert changes in case it was set to `true` beforehand, due to an abnormal initialization sequence.

From *OPL* the transition to *STB*, *opl→stb*, is taken if the `stb_flg` is set to `true` again. This does not have any effect on the `recovery_flg`.

For testing purposes, it is possible to activate *XBIT* from *STB*. However, this is not directly possible from *OPL* due to safety concerns. To activate *XBIT*, transition *stb→xbit* needs to be satisfied. This can be done by setting `xbit_flg` to `true`. In this case, *XBIT* is activated and the corresponding `level1_lgx` value of two is set.

A return to *STB* is possible by satisfying *xbit→stb*. When `xbit_flg` is set to `false`, the transition to *STB* is executed. Due to the same reasons as above a direct change from *XBIT* to *OPL* is not possible.

Table 4.2: *Level 1 - Transition Matrix*

		To		
		OPL	STB	XBIT
From	init	*init→opl*	*init→stb*	n/a
	OPL	-	*opl→stb*	n/a
	STB	*stb→opl*	-	*stb→xbit*
	XBIT	n/a	*xbit→stb*	-

4.3.2 Level 2

The *Level 2* of the SA includes the child modes of *OPL*. They consist of a mode for the EP and the FO, *OPL_External-Pilot* (*OPL_EP*) and *OPL_Flight-Operator* (*OPL_FO*), as well as their respective link loss modes, *OPL_External-Pilot Link Loss* (*OPL_EPLL*) and *OPL_Flight-Operator Link Loss* (*OPL_FOLL*).

The *Stateflow* state chart, with those four modes, is depicted in Figure 4.13 and the necessary inputs are listed in Table 4.3. It can be seen, that only one output is calculated in this state machine, which reflects the current state in enumerations from one to four. This output is set to the respective enumeration on entering the state, by the transition actions located before the states. An overview of all possible transitions is shown in the transition matrix in Table 4.4. Those are also visible in Figure 4.13 and will be discussed in the following.

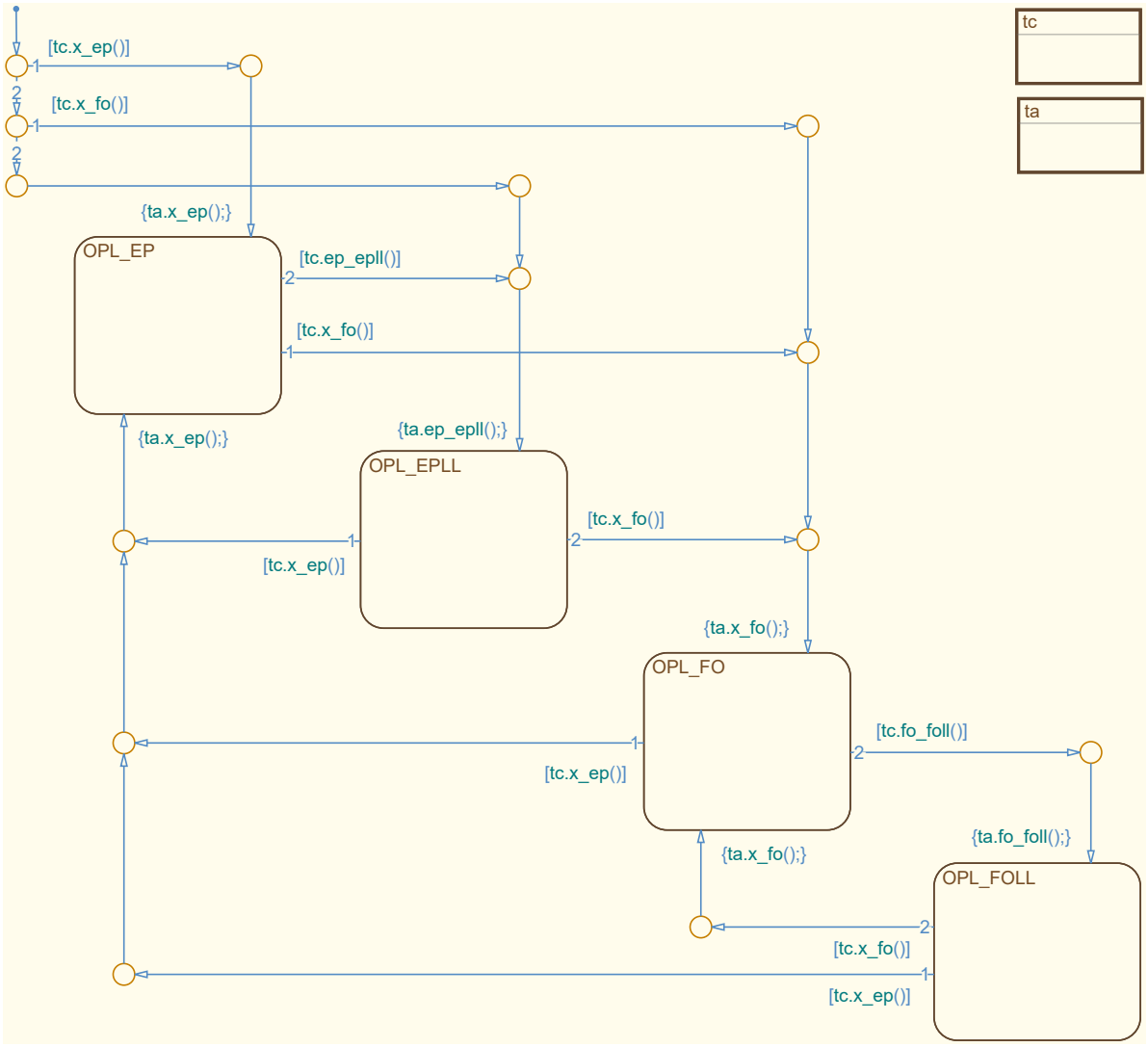


Figure 4.13: *Level 2 (OPL) - State Machine*

Table 4.3: *Level 2 (OPL) - Interface*

Name	Direction	Datatype	Range
`fo_rfg`	input	boolean	0, 1
`ep_rfg`	input	boolean	0, 1
`foll_flg`	input	boolean	0, 1
`epll_flg`	input	boolean	0, 1
`atol_mode_lgx`	input	enum	40, 41, 42, 43, 44
`atol_takeoff_compl_flg`	input	boolean	0, 1
`atol_toabort_compl_flg`	input	boolean	0, 1
`atol_landing_compl_flg`	input	boolean	0, 1
`atol_goaround_compl_flg`	input	boolean	0, 1
`level2_lgx`	output	enum	0, 1, 2, 3, 4

This chart will be active, whenever the *Level 1* mode *OPL* is activated. This also means the initialization, e.g. default transition, is performed again and the state machine doesn't continue from its last known state. When entering any mode the value for `level2_lgx` is set within the respective transition action.

Throughout the state chart, consistent prioritization is used. Whenever link loss modes are considered, their priority is the lowest. This is due to the unlikely event of a simultaneous link loss and control takeover by either EP or FO. In this case, the human input is prioritized. An example situation would be if the aircraft is controlled by the EP and a link loss occurs. If at this exact moment the FO wants to take over control as well, the state machine will go to *OPL_FO* instead of *OPL_EPLL*.

Additionally, the EP always has a higher priority than the FO, solving another unlikely event of a simultaneous command request from both users from a link loss mode. This could be the case if a link loss has occurred and the aircraft is flying in *OPL_FOLL* and returning to the base. When the EP and FO want to take over control at the same moment, the state machine will go to *OPL_EP* instead of *OPL_FO*. The main reason for this being, the inability of the EP to monitor the current mode of the FCC. Since this is displayed to the FO, a second command request can be made if necessary, which will finally trigger the automation to change to *OPL_FO*.

The default transition has three possible mode outcomes, namely *OPL_EP*, *OPL_FO*, or *OPL_EPLL*. This decision depends only on `ep_rfg` and `fo_rfg`. Both flags originate from a switch on the remote control of the EP or buttons in the GCS of the FO. The state of the button or switch is sent to the aircraft, and the SA uses external decision logic to detect a rising edge. If the rising edge occurs at the moment the state chart becomes active, *init→ep* or *init→fo* is executed. However, in most cases, the rising edge does not occur in the same time step as the initialization and therefore *OPL_EPLL* is activated by *init→epll*.

Table 4.4: *Level 2 (OPL) - Transition Matrix*

		To			
		EP	EPLL	FO	FOLL
From	init	*init→ep*	*init→epll*	*init→fo*	n/a
	EP	-	*ep→epll*	*ep→fo*	n/a
	EPLL	*epll→ep*	-	*epll→fo*	n/a
	FO	*fo→ep*	n/a	-	*fo→foll*
	FOLL	*foll→ep*	n/a	*foll→fo*	-

During the startup of the aircraft, *OPL_EPLL* is used as a transition state to normal operation since the link cannot be established beforehand, even though no "real" link loss has occurred. This mode is used since its ground behavior (control surfaces centered, thrust idle, brakes applied) is required.

From *OPL_EPLL* the same transition conditions are used to either activate *OPL_EP* or *OPL_FO* via *epll→ep* or *epll→fo* respectively. This means the EP or FO has to actively request command authority and therefore cannot get into an unprepared control situation.

If *OPL_EP* is active the FO can request control via *ep→fo*, which is only restricted by the `fo_rfg`. However, if a link loss takes place *OPL_EPLL* is automatically activated via *ep→epll*. This transition is guarded by the `epll_flg`. Depending on the airborne status of the aircraft different timeouts are used for the data links. However, this consolidation is moved to external logic combining integrity information of the data link with the airborne status of the aircraft, which helps to reduce complexity within the state machine.

Assuming the FO requested control by activating the button in the GCS, *ep→fo* is executed and *OPL_FO* is active. From this state, the EP can gain control by the same means as discussed above and utilizing *fo→ep*. However, if a link loss is detected *fo→foll* is taken and *OPL_FOLL* is automatically activated. In a similar way, as with *OPL_EPLL*, this decision is based on `foll_flg`, which in turn is calculated by external logic using integrity information of the data link and the airborne status of the aircraft.

There is, however, one exception, where *OPL_FOLL* is not automatically activated. This is the case when the child mode *OPL_FO_ATOL* of *OPL_FO* is active. *OPL_FOLL* is inhibited during this phase of the flight because *OPL_FO_ATOL* has its own specialized link loss modes that are specifically designed to deal with the close proximity to the ground. This decision is based on the enumerated `atol_mode_lgx` in combination with the four flags `atol_takeoff_compl_flg`, `atol_toabort_compl_flg`, `atol_landing_compl_flg` and `atol_goaround_compl_flg`.

As with *OPL_EPLL*, an active command from either the EP or FO is required for a transition. Depending on `ep_rfg` and `fo_rfg` the respective transition, *foll→ep* or *foll→fo* is taken to either *OPL_EP* or *OPL_FO*.

4.3.3 Level 3

The four modes on *Level 2* all have child modes on *Level 3*. Those parent modes on *Level 2* are listed with their respective child modes in the following.

- Operational - EP - GND/AIR
- Operational - FO - PARK/MLC/HLC/RTB/ATOL
- Operational - EP Link Loss - GND/AIR
- Operational - FO Link Loss - GND/BANK/HDG/GPS

The following transition conditions and action of modes on *Level 3* are divided with respect to their *Level 2* parent mode to give a better understanding.

4.3.3.1 Operational - EP - Ground/Airborne

The *Level 2* mode *OPL_EP* is a parent to two child modes on *Level 3*. They are *OPL_EP_Air* (*OPL_EP_AIR*) and *OPL_EP_Ground* (*OPL_EP_GND*). Their transition conditions and actions are described in the following.

The state chart is depicted in Figure 4.14, the interface is listed in Table 4.5 and the transition matrix is shown in Table 4.6.

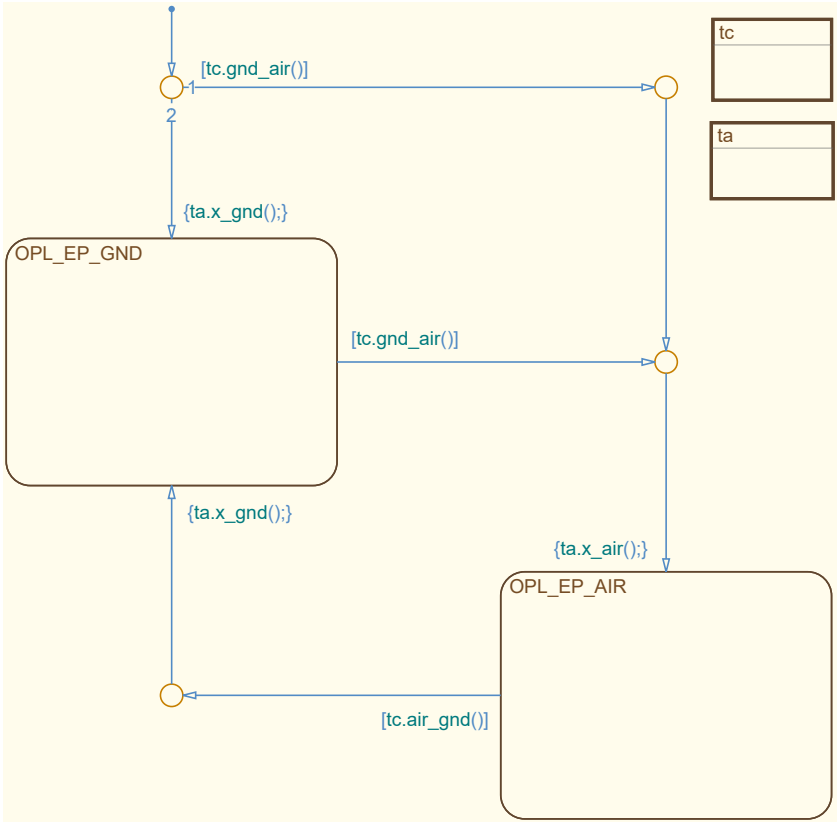


Figure 4.14: *Level 3 (OPL-EP) - State Machine*

Table 4.5: *Level 3 (OPL-EP) - Interface*

Name	Direction	Datatype	Range
`airborne_lgx`	input	enum	0, 1, 2, 3
`level3_lgx`	output	enum	20, 21

All transitions of this state machine are based on `airborne_lgx`, which describes the airborne status of the aircraft in four increments. The definition of the enumerated values of `airborne_lgx` are listed in the following.

- 0 - Ground (nose wheel and main wheels on the ground)
- 1 - Ground to air transition (nose wheel airborne, main wheels still on the ground)
- 2 - Air to ground transition (main wheels on the ground, nose wheel still airborne)
- 3 - Airborne (nose wheel and main wheels airborne)

The transition states (from ground to air and vice versa) are necessary for some controllers, which need to switch to another control law before the aircraft is completely airborne (or on the ground respectively). A simplified example implementation is given in brackets. While the transition states basically have the same definition, the differentiation arises from their source state.

During the initialization of the state chart, the transition *init→air* is checked first. If the aircraft is completely airborne *OPL_EP_AIR* is activated and the respective `level3_lgx` is set. Otherwise, *OPL_EP_GND* is active. The transition from *gnd→air* is guarded by the same statement as *OPL_EP_AIR*. However, the opposite transition from *OPL_EP_AIR* to *OPL_EP_GND*, *air→gnd*, is executed if `airborne_lgx` is "ground" or "ground to air transition". In the example implementation, that means that *air→gnd* is activated as soon as the two main wheels touch the ground.

The transitions used in the link loss case, which are presented in Subsubsection 4.3.3.3, are very similar. They only differ in the fact, that the transition to *OPL_EPLL_AIR*, is executed in the "airborne" as well as "ground to air transition" state. This is due to the different control loops and strategies used in both modes, which are explained in Subsection 4.2.2 and Subsection 4.2.3.

Table 4.6: *Level 3 (OPL-EP) - Transition Matrix*

		To	
		GND	AIR
From	init	*init→gnd*	*init→air*
	GND	-	*gnd→air*
	AIR	*air→gnd*	-

4.3.3.2 Operational - FO - PARK/MLC/HLC/RTB/ATOL

The *Level 3* child modes of *OPL_FO* consist of *OPL_FO_Parking* (*OPL_FO_PARK*), *OPL_FO_Medium-Level-Control* (*OPL_FO_MLC*), *OPL_FO_High-Level-Control* (*OPL_FO_HLC*), *OPL_FO_Return-to-Base* (*OPL_FO_RTB*), and *OPL_FO_Automatic Takeoff and Landing* (*OPL_FO_ATOL*), which is explained in the following. Those are the main operating modes for the FO and combined with the hidden mode (*OPL_FO_ATOL-Unconfirmed* (*OPL_FO_ATUC*)) this leads to the most complex state chart of the SA.

The state chart, which is handling those operating modes is depicted in Figure 4.15. Its necessary interface is listed in Table 4.7 and the possible transitions between the modes are shown in Table 4.8. It mostly uses `opl_mode_lgx`, which is the command for a specific operating mode by the FO. The operating mode in the previous time step is preserved and also used as input via `last_mode_lgx`. Additionally, integrity information about the position of the aircraft, `nav_posOK_flg`, and its airborne status `airborne_lgx` are utilized. This level is tightly connected to ATOL and therefore also needs information from that flight control module. Those consist of the current mode, `atol_mode_lgx`, and availability information for automatic takeoff and landing, `atol_takeoff_avbl_flg` and `atol_landing_avbl_flg`. The outputs consist of the *Level 3* operation mode representation, `level3_lgx`, and a communication request for ATOL, `atuc_flg`.

The chart will be active when the FO is in control. As with the other state charts, it is also re-initialized via the default transition whenever the FO takes over the command.

At the initialization of the state chart, e.g. whenever the FO starts to control the aircraft, four of the six states are possible. Firstly, the airborne status of the aircraft is checked using `airborne_lgx`. When the aircraft is still on the ground, *init→park* is executed and *OPL_FO_PARK* is activated. This will be the case for a normal mission, where the FO takes over control before takeoff. If the aircraft is airborne, one of the other default transitions is used. If the FO commands *OPL_FO_RTB*, while requesting control of the aircraft, *init→rtb* is utilized to activate this mode. However, this is only executed if a valid position of the aircraft is available. In a similar way, *OPL_FO_HLC* is activated via *init→hlc* if requested by the FO and a position is available. Those guards are implemented by checking `nav_posOK_flg`, which is necessary because the activation of both *OPL_FO_RTB* and *OPL_FO_HLC* requires a valid position of the aircraft as explained in Subsubsection 4.2.3.3. If none of these conditions can be satisfied the transition *init→mlc* is executed and *OPL_FO_MLC* is activated. This mode can be viewed as the default activation mode of the FO since it is activated if it is specifically requested, if no mode command is received or if another mode, which is commanded by the FO, cannot be activated due to malfunctions.

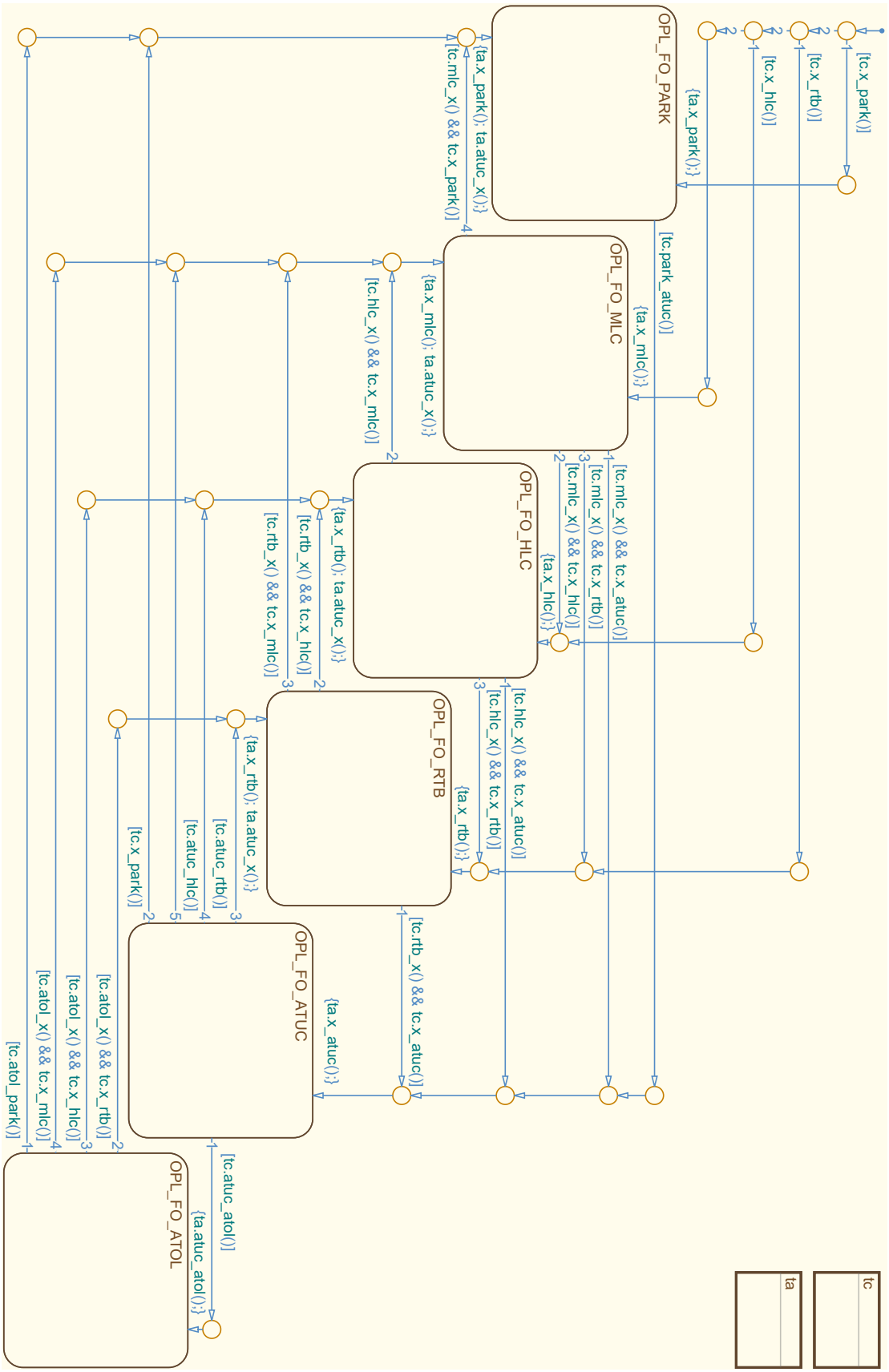


Figure 4.15: *Level 3 (OPL-FO) - State Machine*

Table 4.7: *Level 3 (OPL-FO) - Interface*

Name	Direction	Datatype	Range
opl_mode_lgx	input	enum	20, 21, 22, 26, 40, 42
nav_posOK_flg	input	boolean	0, 1
airborne_lgx	input	enum	0, 1, 2, 3
atol_takeoff_avbl_flg	input	boolean	0, 1
atol_landing_avbl_flg	input	boolean	0, 1
atol_mode_lgx	input	enum	40, 41, 42, 43, 44
last_mode_lgx	input	enum	0, 1
level3_lgx	output	enum	0, 1, 2, 3, 4, 5
atuc_flg	output	boolean	0, 1

During a normal activation by the FO, when the aircraft is on the ground, *OPL_FO_PARK* is activated. While this mode has various entry conditions, it has only one exit transition. This is *park→atuc* to *OPL_FO_ATUC*, which is executed when automatic takeoff is commanded by the FO via opl_mode_lgx and ATOL reports it as available via atol_takeoff_avbl_flg. This results in a setting of atuc_flg but in no change of level3_lgx. Therefore this state is hidden to the operator and only used as an intermediate step.

OPL_FO_ATUC is a hidden mode, with respect to the FO and therefore was not explained in the previous section. Due to the structure of the flight control modules, it is necessary to create a verified transition to automatic takeoff, as well as to landing.

The atuc_flg is used by the SA to permit ATOL to switch into the respective mode, commanded by the FO. Due to the FCC internal execution order of the different flight control modules, ATOL receives this command in the same time step SA transitioned into *OPL_FO_ATUC*. The ATOL module then transitions into the takeoff mode and reports this to the SA via atol_mode_lgx in the next time step. This is then used in *atuc→atol* to switch to *OPL_FO_ATOL*. Depending on the reported mode, in this case, automatic takeoff, the level3_lgx is set accordingly. If this mode information is not received in the next time step the transition *atuc→park* is executed, if the aircraft is still on the ground, and *OPL_FO_PARK* is activated again.

If positive feedback is received from ATOL, the aircraft performs a fully automatic takeoff and initial climb. When the aircraft is airborne three transitions are possible. If *OPL_FO_HLC* is commanded by the FO (opl_mode_lgx) and the GPS position is available (nav_posOK_flg) the transition *atol→hlc* is executed and *OPL_FO_HLC* is activated. In the same way, *atol→rtb* can be used and *OPL_FO_RTB* is activated. The transition *atol→mlc* is not guarded by the position availability because *OPL_FO_MLC* doesn't need a valid position of the aircraft. This mode is also activated if *OPL_FO_HLC* or *OPL_FO_RTB* is requested by the pilot, but a valid position is not available.

Table 4.8: *Level 3 (OPL-FO) - Transition Matrix*

		To					
		PARK	MLC	HLC	RTB	ATUC	ATOL
From	init	*init→park*	*init→mlc*	*init→hlc*	*init→rtb*	n/a	n/a
	PARK	-	n/a	n/a	n/a	*park→atuc*	n/a
	MLC	*mlc→park*	-	*mlc→hlc*	*mlc→rtb*	*mlc→atuc*	n/a
	HLC	n/a	*hlc→mlc*	-	*hlc→rtb*	*hlc→atuc*	n/a
	RTB	n/a	*rtb→mlc*	*rtb→hlc*	-	*rtb→atuc*	n/a
	ATUC	*atuc→park*	*atuc→mlc*	*atuc→hlc*	*atuc→rtb*	-	*atuc→atol*
	ATOL	*atol→park*	*atol→mlc*	*atol→hlc*	*atol→rtb*	n/a	-

The three operating modes that are used by the FO during non-terminal flight phases are *OPL_FO_MLC*, *OPL_FO_HLC*, and *OPL_FO_RTB*. The state machine will use one of the six transitions *mlc→hlc*, *mlc→rtb*, *hlc→mlc*, *hlc→rtb*, *rtb→mlc*, or *rtb→hlc* to switch between those modes if commanded by the FO. In the case of activating either *OPL_FO_HLC* or *OPL_FO_RTB*, this requires a valid GPS position. A special case applies to the transition *hlc→mlc*. It is triggered automatically if the position information is lost during *OPL_FO_HLC*. This does not apply to *rtb→mlc*, due to the available *Level 4* modes of *OPL_FO_RTB*.

From all three non-terminal operating modes, *OPL_FO_MLC*, *OPL_FO_HLC*, and *OPL_FO_RTB*, a fully automatic landing can be commanded by the FO. If commanded via `opl_mode_lgx` and reported as available by the ATOL via `atol_landing_avbl_flg` one of the transitions is executed. Depending on the previously active mode *mlc→atuc*, *hlc→atuc*, or *rtb→atuc* is executed and *OPL_FO_ATUC* is activated. This is reported to ATOL via `atuc_flg` and fed back to the SA via `atol_mode_lgx` in the next time step.

If the feedback is received from ATOL in the next time step the transition *atuc→atol* is executed and *OPL_FO_ATOL* is activated. Within this transition, `level3_lgx` is set to the respective value for automatic landing. In case that the correct feedback is not received, the transition to the previous mode is executed. Depending on `last_mode_lgx`, which preserves this information, either *atuc→mlc*, *atuc→hlc*, or *atuc→rtb* is executed and the respective mode is activated.

After the automatic landing, the transition *atol→park* can be used by the FO to activate *OPL_FO_PARK*. This is guarded by the `opl_mode_lgx` command and the airborne status of the aircraft, `airborne_lgx`.

For an emergency landing by the FO without ATOL, a possible direct transition from *OPL_FO_MLC* to *OPL_FO_PARK* is available. The transition *mlc→park* is executed if commanded by the FO via `opl_mode_lgx`, and the aircraft is on the ground, which is analyzed using `airborne_lgx`.

4.3.3.3 Operational - EP Link Loss - Ground/Airborne

The *OPL_EPLL* child modes are very similar to the nominal ones of *OPL_EP* (c.f. Subsubsection 4.3.3.1). Therefore, further explanation is omitted here, and only the respective chart is depicted in Figure 4.16, the interface table is listed in Table 4.9, and the transition matrix is shown in Table 4.10.

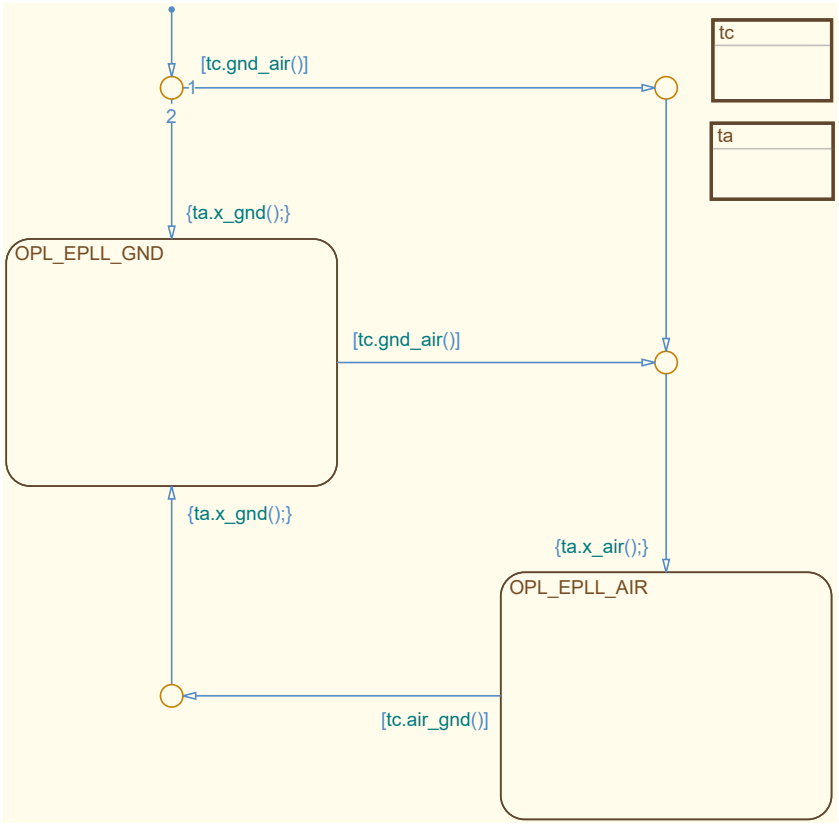


Figure 4.16: *Level 3 (OPL-EPLL) - State Machine*

Table 4.9: *Level 3 (OPL-EPLL) - Interface*

Name	Direction	Datatype	Range
`airborne_lgx`	input	enum	0, 1, 2, 3
`level3_lgx`	output	enum	30, 31

Table 4.10: *Level 3 (OPL-EPLL) - Transition Matrix*

		To	
		GND	AIR
From	init	*init→gnd*	*init→air*
	GND	-	*gnd→air*
	AIR	*air→gnd*	-

4.3.3.4 Operational - FO Link Loss - GND/BANK/HDG/GPS

There are four *Level 3* child modes of *OPL_FOLL*, which are used to distinguish between different FO link loss situations. Those are *OPL_FOLL_Ground* (*OPL_FOLL_GND*), *OPL_FOLL_Bank Angle* (*OPL_FOLL_BANK*), *OPL_FOLL_Heading* (*OPL_FOLL_HDG*), and *OPL_FOLL_GPS* (*OPL_FOLL_GPS*).

The *Stateflow* state machine, with its four states for handling the link loss, is depicted in Figure 4.17. From the interface table, Table 4.11, it can be seen that all transition conditions can only rely on the validity of the position of the aircraft, `nav_posOK_flg`, and the airborne status, `airborne_lgx`. Additionally, all available transitions are shown in the transition matrix, Table 4.12. However, due to the limited interactions during a link loss, only a very small number of transitions are available and the majority of transitions are inhibited. Whenever a state is entered, the respective value for `level3_lgx` is set.

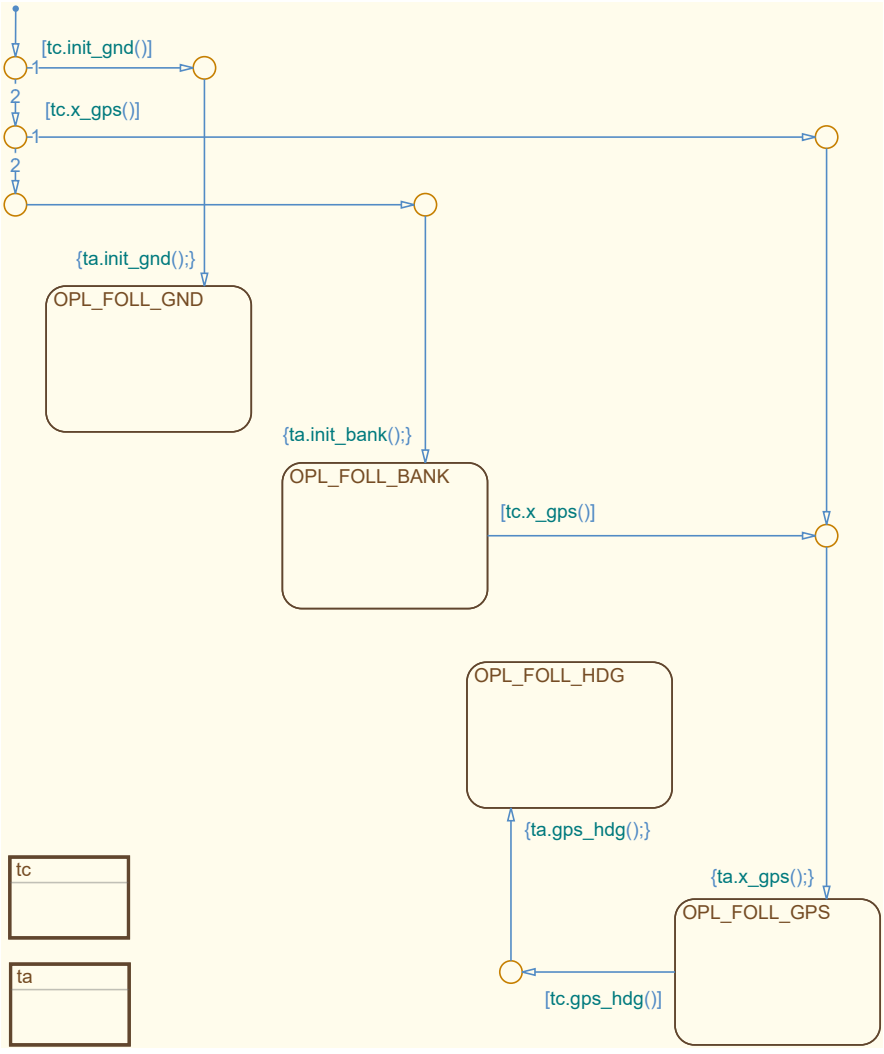


Figure 4.17: *Level 3 (OPL-FOLL) - State Machine*

Table 4.11: *Level 3 (OPL-FOLL) - Interface*

Name	**Direction**	**Datatype**	**Range**
`nav_posOK_flg`	input	boolean	0, 1
`airborne_lgx`	input	enum	0, 1, 2, 3
`level3_lgx`	output	enum	10, 11, 12, 13

There are three possible transitions during initialization. The highest priority one is *init→gnd*, which leads to *OPL_FOLL_GND*. It is guarded by `airborne_lgx` and only executed if the aircraft is still on the ground. This state is not used during "normal" operation but exists only for the case of a link loss during *OPL_FO_PARK*. It has no exit condition and is, therefore, a terminal state.

A link loss, while being on the ground can otherwise only occur during takeoff and landing and is in this case handled by ATOL. If the aircraft is airborne when the link loss occurs and a valid position can be calculated (`nav_posOK_flg` is `true`), *init→gps* is executed and *OPL_FOLL_GPS* is activated. Otherwise, *init→bank*, which has no condition, is executed and *OPL_FOLL_BANK* is activated.

As discussed previously, if the link loss occurs, without a valid position of the aircraft, *OPL_FOLL_BANK* is activated. From this mode, there is only one transition to *OPL_FOLL_GPS* via *bank→gps*. It is executed in case the GPS reception can be restored. The slow upward spiral during *OPL_FOLL_BANK* increases the probability of a valid GPS signal and therefore also the chance of the transition to *OPL_FOLL_GPS*.

Once in *OPL_FOLL_GPS*, there is a possible transition to *OPL_FOLL_HDG*. The transition *gps→hdg* is executed if an (additional) GPS loss occurs. The state *OPL_FOLL_HDG* is also a terminal state, which has no exit conditions. The reasons for this behavior are explained in Subsubsection 4.2.3.4. However, if the GPS position can be obtained again the heading used in this mode is updated and leads to the homing behavior depicted in Figure 4.8 as "HDG with updates".

Table 4.12: *Level 3 (OPL-FOLL) - Transition Matrix*

		To			
		GND	BANK	HDG	GPS
From	init	*init→gnd*	*init→bank*	n/a	*init→gps*
	GND	-	n/a	n/a	n/a
	BANK	n/a	-	n/a	*bank→gps*
	HDG	n/a	n/a	-	n/a
	GPS	n/a	n/a	*gps→hdg*	-

4.3.4 Level 4

There are only a few modes on *Level 4*, which are child modes of *OPL_FO_RTB* and therefore belong to the SA. The others are part of ATOL and are omitted here.

4.3.4.1 Operational - FO - RTB - GPS/HDG

The *Level 4* state machine is depicted in Figure 4.18, its in- and outputs are listed in Table 4.13 and the transition matrix is shown in Table 4.14.

The activation of the *Level 3* parent mode *OPL_FO_RTB* is only possible with a valid GPS signal. Therefore the only transition during initialization on this level is *init→gps*. This leads to *OPL_FO_RTB_GPS* (*OPL_FO_RTB_GPS*) being active and the respective setting of `level4_lgx`.

If a GPS loss occurs (`nav_posOK_flg` is `false`), the transition *gps→hdg* is executed and *OPL_FO_RTB_Heading* (*OPL_FO_RTB_HDG*) is activated.

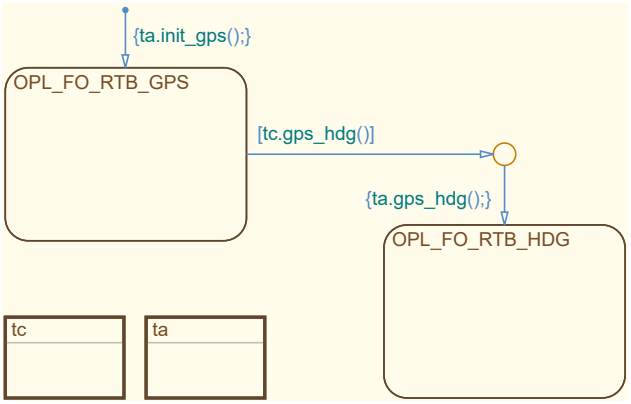


Figure 4.18: *Level 4 (OPL-FO-RTB) - State Machine*

Table 4.13: *Level 4 (OPL-FO-RTB) - Interface*

Name	Direction	Datatype	Range
`nav_posOK_flg`	input	enum	0, 1
`level4_lgx`	output	enum	0, 1

Table 4.14: *Level 4 (OPL-FO-RTB) - Transition Matrix*

		To	
		HDG	GPS
From	init	n/a	*init→gps*
	HDG	-	n/a
	GPS	*gps→hdg*	-

4.4 Loiter Automation

Multiple modes in the SA are designed as contingency procedures to mitigate the effects of mission changes and/or malfunctions and to guarantee a continuous automatic operation. The *Loiter Automation (LA)* is, due to its complexity, implemented in an encapsulated submodule. Together with the other contingency modes, it represents contribution *C2.3 - Automatic operational and malfunction contingency procedures for continuous operation in non-nominal circumstances*.

It is implemented in parallel to the *Level 3* state machine of *OPL_FO*, which is described in Subsubsection 4.2.3.3. The loiter modes are therefore generally available in all FO-controlled modes of the SA. However, depending on the *Level 3* mode, some loiter modes or entering a holding pattern in general, might be inhibited by the LA.

The ability to "pause" the mission is necessary for both, UAV and OPV. Loiter defines a flight phase in which the aircraft "holds" its current position. The maneuver is normally conducted in a circular or racetrack-pattern. In civil aviation, it normally occurs shortly before landing, while the aircraft waits for the landing clearance. In military missions, loiter can be used during a reconnaissance mission in which the aircraft flies over its target in circles. The interface for the FO in the GCS is designed to be very easy but versatile. It uses only a boolean command and an integer-represented mode response.

4.4.1 Loiter Modes

For *SAGITTA* and the *DA 42*, which are used as demonstration platforms for the FCSA, loiter can be commanded by the FO or it is automatically activated in certain situations. Loiter can be entered in three different ways, which are listed in the following.

- Loiter can be triggered by the FO
- Loiter is automatically activated when a loiter waypoint is passed
- Loiter is automatically activated when the end of the flight plan is reached

The six modes, used in the main state machine of the LA, are listed below. How they are used to create the loiter functionality is explained in the following.

- Off
- Bank
- GPS
- Unconfirmed
- Will Exit
- End of Flight Plan

However, beforehand a short overview of the interface, of the LA, is presented, to get a better comprehension of the environment in which the modes are used.

The interface of the LA consists only of one switch or button in the GCS of the FO and a response variable of the currently active mode. Whenever the FO wants to start a loiter maneuver the switch needs to be activated. This is sent to the aircraft via the data link and used by the LA to start the respective maneuver. This is generally independent of the currently active mode of the SA. However, in certain modes, like automatic takeoff or landing, loitering is not permitted.

To deactivate the loiter pattern, the switch needs to be turned off. However, depending on the currently active mode of the LA, the flight is not immediately resumed. This behavior and the necessary reaction of the FO are explained in the following.

4.4.1.1 Off

The default mode for the state machine is *Loiter_OFF* (*LTR_OFF*). It is used for all non-loiter flight phases or modes in which loiter cannot be activated.

4.4.1.2 Bank

One of the immediate loiter patterns that can be triggered by the FO is *Loiter_Bank* (*LTR_BANK*). As the name suggest the loiter pattern in this mode is based on a constant and predefined bank-angle (e.g. 30°). This is used as a loiter pattern if the *Level 3* mode is *OPL_FO_MLC*. In perfect conditions, this leads to a circular pattern that passes the point of activation on every round. However, due to wind or other disturbances, the path of the aircraft with respect to the ground can change. Since this is comparable to the normal *OPL_FO_MLC* mode and is expected by the FO, it is not mitigated by the SA.

If commanded by the FO, the same control loops that are used for *OPL_FO_MLC*, are utilized with a fixed lateral command. Other command values like altitude and speed are not modified and can be controlled by the FO. An exit from this holding pattern is executed immediately if commanded by the FO and the command for the AFCS is switched back from bank-angle to e.g. heading mode.

4.4.1.3 GPS

The mode *Loiter_GPS* (*LTR_GPS*) is the second possibility for the FO to trigger an immediate loiter. It is based on the GPS position of the aircraft and used if *OPL_FO_HLC* is active. It utilizes a special mode of the TG module, in which a loiter circle is calculated based on a predefined turn rate (e.g. $3°/s$). With the current speed of the aircraft, the TG and TC are able to track the loiter circle even in presence of disturbances like the wind.

Superposition modes like *Altitude Superposition* and *Speed Superposition* are part of *OPL_FO_HLC* and can still be used by the FO during *LTR_GPS*. This mode is also eventually used at a loiter waypoint. The process of entering and exiting such a holding pattern automatically at a loiter waypoint is explained in the following.

4.4.1.4 Unconfirmed

A loiter pattern can also be entered at a specific waypoint, if the parameter for loitering, of that point, is enabled. In this case, the holding pattern is automatically entered and the mode *Loiter_Unconfirmed* (*LTR_UNCO*) is activated. This, however, results in an inconsistency between the aircraft and the GCS, because the aircraft entered the loiter pattern without command, as indicated by the mode name.

Since this is planned beforehand, this is in accordance with the flight plan and expected by the FO. In this situation, the aircraft is loitering, while the FO switch, which is commanding loiter, is still off. However, this mismatch can easily be identified in the GCS and can even be resolved automatically if soft buttons are used. The mode is maintained until the command is adjusted by the FO, which is then confirmed by switching to the normal GPS-based loiter mode, *LTR_GPS*.

4.4.1.5 Will Exit

When the aircraft is loitering in *LTR_GPS* this can be either due to a direct command from the FO or because of a loiter waypoint. In this situation, the exit from *LTR_GPS* needs to be executed in a predictable and deterministic way so that the planned route can be continued. It can be commanded by the FO by deactivating the command switch for loitering, however, the loiter pattern is not immediately exited.

In this case, *Loiter_Will Exit* (*LTR_WIEX*) is used until the aircraft reaches the entry point of the loiter pattern. By making sure holding rounds are completed, the correct and possible continuation of the flight plan is ensured. When passing the entry/exit point the mode is automatically disabled and *LTR_OFF* is activated, thus exiting the loiter pattern and continuing the normal flight plan.

4.4.1.6 End of Flight Plan

The flight plan of the aircraft can consist of many individual waypoints. However, this number has to be finite, and therefore a possibility of reaching the last waypoint exists. In this situation, the aircraft automatically enters a loiter pattern, because no other reasonable alternative exists. This special mode for a GPS-based loiter is indicated by activating *Loiter_End-of-Flight-Plan* (*LTR_EoFP*).

The mode is activated immediately, without acknowledgment by the FO in the GCS, because there is no possible continuation of a flight plan afterward. Since no consequent waypoints exist, no exit procedure, like during *LTR_GPS*, exists either. Therefore the loiter pattern and the mode are exited immediately if another control mode or flight plan is selected by the FO.

4.4.2 Transition Conditions and Actions

The LA, which is implemented on *Level 3* of the SA uses the six modes *Loiter_OFF* (*LTR_OFF*), *Loiter_Bank* (*LTR_BANK*), *Loiter_GPS* (*LTR_GPS*), *Loiter_Unconfirmed* (*LTR_UNCO*), *Loiter_Will Exit* (*LTR_WIEX*), and *Loiter_End-of-Flight-Plan* (*LTR_EoFP*) to realize the loiter functionality as described in Subsection 4.4.1.

The *Stateflow* state machine is depicted in Figure 4.19. An overview of the possible transitions is given in Table 4.16. The relevant interface variables are listed in Table 4.15. Those include the mode of the *OPL_FO* main state machine, `level3_lgx`, and an externally calculated change, `level3_lgx_cfg`, which are used together with the command of the FO, `loiter_cmd_flg`, to distinguish between different modes and trigger automatic changes. Furthermore, `loiter_flg` is used to command the start of the loiter to the TG module and `loiter_lgx` is used as feedback.

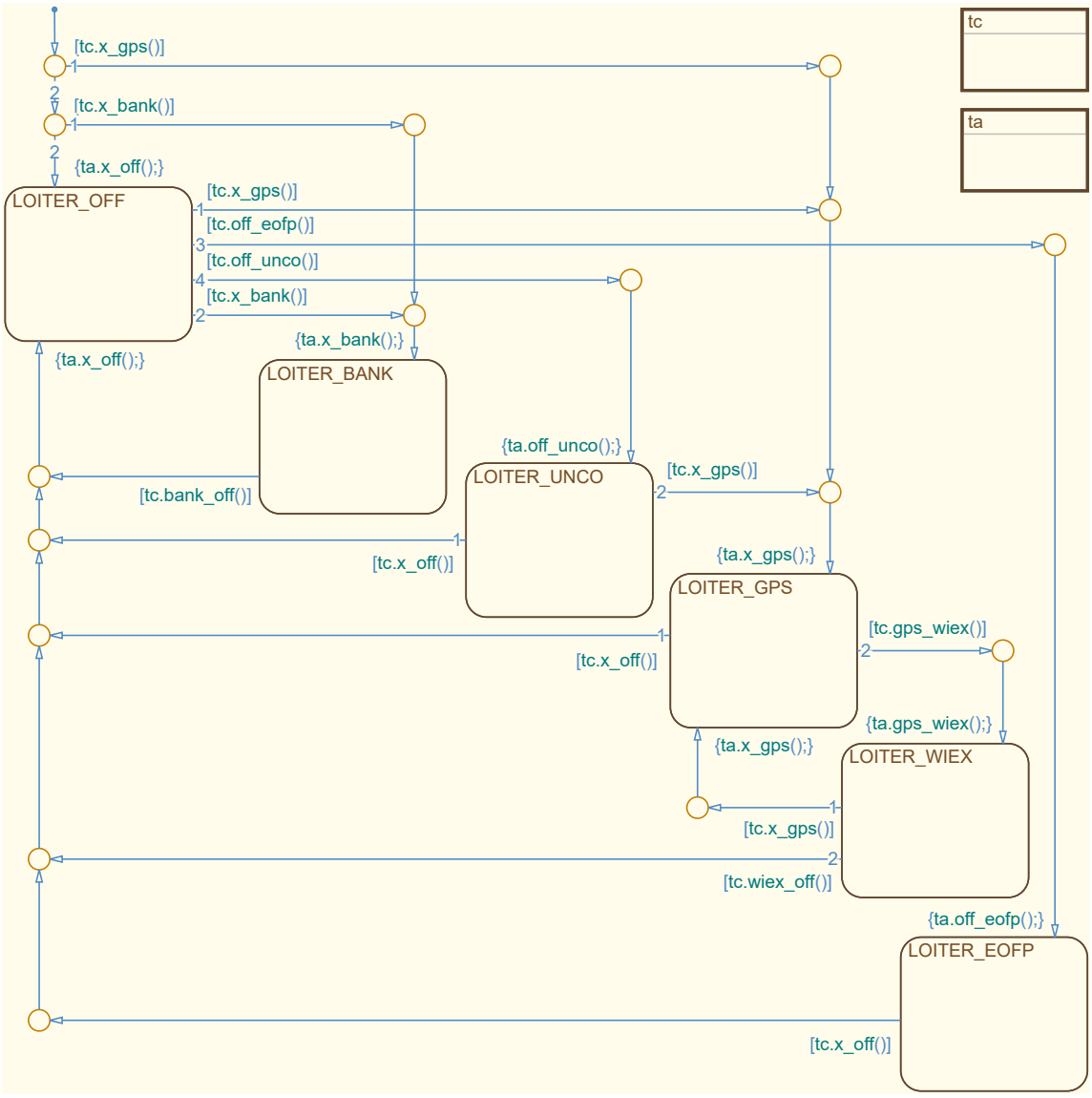


Figure 4.19: *Level 3 (OPL-FO) Loiter - State Machine*

Table 4.15: *Level 3 (OPL-FO) Loiter - Interface*

Name	Direction	Datatype	Range
`level3_lgx`	input	enum	0, 1, 2, 3, 4, 5
`level3_lgx_cfg`	input	boolean	0, 1
`loiter_cmd_flg`	input	boolean	0, 1
`trajgen_loiter_flg`	input	boolean	0, 1
`trajgen_loiter_rfg`	input	boolean	0, 1
`trajgen_eof_flg`	input	boolean	0, 1
`enroute_list_cfg`	input	boolean	0, 1
`loiter_lgx`	output	enum	0, 1, 2, 3, 4, 5
`loiter_flg`	output	boolean	0, 1

Additionally, information from the TG module is used to coordinate GPS-dependent mode changes. Those variables are `trajgen_loiter_flg`, `trajgen_loiter_rfg`, and `trajgen_eof_flg`.

During the initialization of the state chart, three transitions and therefore three initial modes are possible. Those are *LTR_GPS*, *LTR_BANK*, and *LTR_OFF* with the latter being the default one without any condition. Therefore, the transition *init→off* is executed if no previous condition is met during the first time step of execution. This will most likely be the case since otherwise `loiter_cmd_flg` had to be set by the FO before requesting control at all.

So normally, the default transition activates *LTR_OFF* which sets the respective value for `loiter_lgx` and additionally sets the `loiter_flg` to `false`, which is used as a command interface to the TG module. However unlikely, the transition *init→gps* is executed if `loiter_cmd_flg` is set during initialization and *OPL_FO_HLC* is active. Similarly, *init→bank* is executed if `loiter_cmd_flg` is set to `true` and *OPL_FO_MLC* is active.

In normal operation, *LTR_OFF* is active after a control-takeover by the FO. From there, an immediate bank-angle-based loiter can be commanded by setting `loiter_cmd_flg`, if the corresponding *Level 3* operating mode is *OPL_FO_MLC*. In this case, *off→bank* is executed, *LTR_BANK* is activated, and the respective `loiter_lgx` is set.

The deactivation of *LTR_BANK*, via *bank→off*, is commenced if either the loiter command by the FO is set to `false` or if the *Level 3* operating mode is changed.

From *LTR_OFF*, *LTR_GPS* can be activated in a similar way to *LTR_BANK*. The transition *off→gps* is executed if `loiter_cmd_flg` is set by the FO while being in *OPL_FO_HLC*. In addition to setting the respective `loiter_lgx`, this also sets `loiter_flg` to `true`, which commands the TG module to activate the GPS-based loiter pattern.

Table 4.16: *Level 3 (OPL-FO) Loiter - Transition Matrix*

		To					
		OFF	BANK	UNCO	GPS	WIEX	EOFP
From	init	*init→off*	*init→bank*	n/a	*init→gps*	n/a	n/a
	OFF	-	*off→bank*	*off→unco*	*off→gps*	n/a	*off→eofp*
	BANK	*bank→off*	-	n/a	n/a	n/a	n/a
	UNCO	*unco→off*	n/a	-	*unco→gps*	n/a	n/a
	GPS	*gps→off*	n/a	n/a	-	*gps→wiex*	n/a
	WIEX	*wiex→off*	n/a	n/a	*wiex→gps*	-	n/a
	EOFP	*eofp→off*	n/a	n/a	n/a	n/a	-

The deactivation via transition *gps→off* is, in contrast to *off→bank*, not executed if the `loiter_cmd_flg` is withdrawn. Due to the different requirements for *LTR_GPS*, the direct change to *LTR_OFF* is only executed if the *Level 3* operating mode changes (`level3_lgx_cfg`) or a different waypoint list is selected (`enroute_list_cfg`).

As said, the transition from *LTR_GPS* to *LTR_OFF* cannot be directly commanded by the FO. If however, the exit of *LTR_GPS* is requested from the FO, by setting `loiter_cmd_flg` to `false`, the transition *gps→wiex*, which activates *LTR_WIEX*, is executed. This sets `loiter_flg` to `false`, which instructs the TG to leave the loiter pattern and continue with the next waypoint when the entry point is passed again.

In *LTR_WIEX* the LA waits for the confirmation of the TG module, that the loiter pattern has been completed. If that is reported by setting `trajgen_loiter_flg` to `false`, the transition *wiex→off* is executed. However, it is also executed if the *Level 3* mode is changed (`level3_lgx_cfg`) or another waypoint list is selected by the FO. The request to leave the holding pattern can also be canceled by the FO, by setting `loiter_cmd_flg` to `true`, which triggers *wiex→gps* and activates *LTR_GPS* again.

If the loiter pattern has been completed *LTR_OFF* is activated again and the aircraft follows the waypoint list. If a waypoint is passed that is parameterized to be a loiter point. The transition *off→unco* is executed and *LTR_UNCO* is activated. This transition is triggered by `trajgen_loiter_rfg`, which indicates the independent activation of loitering by the TG, which is reported to the GCS by setting the respective `loiter_lgx`.

The loiter can be confirmed from the FO in the GCS by setting `loiter_cmd_flg`. In this case, *unco→gps* is executed, *LTR_GPS* is activated, indicating a synchronized status of the aircraft and GCS. If however, the *Level 3* mode is changed or another waypoint list is selected, the transition *unco→off* is executed, which activates *LTR_OFF* again.

The mode *LTR_EoFP* is automatically activated if a loiter pattern is entered at the end of a flight plan. This transition *off→eofp* is guarded by `trajgen_loiter_rfg` and `trajgen_eof_flg`, which indicate this situation. The transition *eofp→off*, and therefore the deactivation of *LTR_EoFP*, is executed in a similar way to others if `level3_lgx_cfg` or `enroute_list_cfg` is `true`.

4.5 Injection Switches

The injection switches, that are controlled by the SA and used within the FCSA to control the cascaded control loop are the second part of contribution *C2.1 - Strategy for switchability between various modes on different authority levels, enabling experimental automation.* As depicted in Figure 4.5 there are injection switches prior to each of the major flight control loops and an additional one before the OP.

As an example, the implementation of the *Inner Loop - Switch (IL-SW)* with eight different inputs for various operating modes is shown in Figure 4.20.

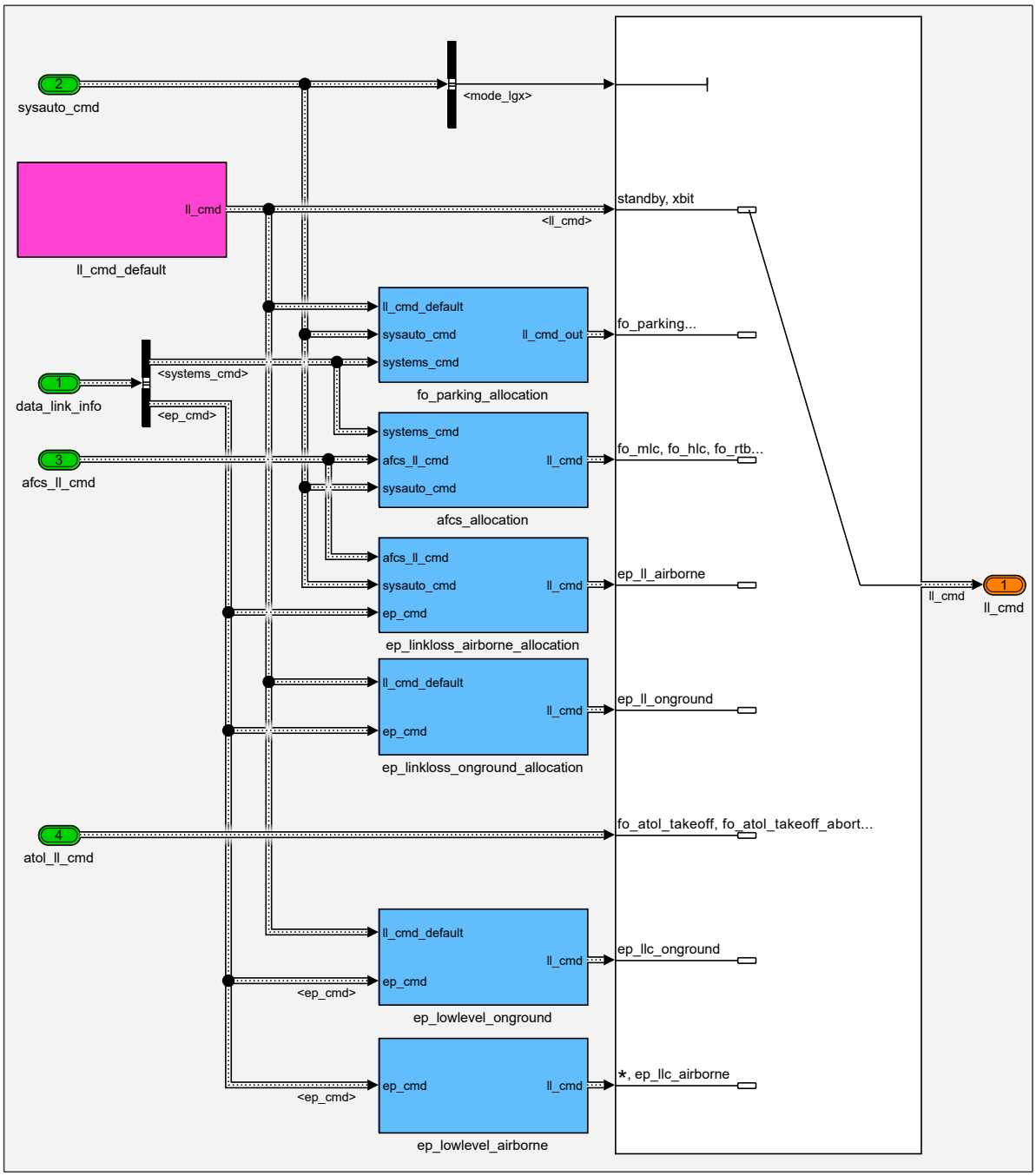


Figure 4.20: *Inner Loop - Switch*

4.5.1 Trajectory Generation - Switch

An overview of the different switch positions of the *Trajectory Generation - Switch (TG-SW)* is shown in Table 4.17. Its used positions are based on the `mode_lgx` calculated by the SA. The first position is used for various modes that do not need the TG module. The input is therefore overwritten with default values, which deactivate the TG. The second position is used for all automatic takeoff- and landing-related operating modes, in which the TG is controlled by the ATOL module. The third position is used for all modes that are controlled by the SA and utilize the TG module.

Table 4.17: *Trajectory Generation - Switch*

Position	**TG - Switch based on `mode_lgx`**
1	*STB*, *XBIT*, *OPL_EP_GND*, *OPL_EP_AIR*, *OPL_EPLL_GND*, *OPL_EPLL_AIR*, *OPL_FO_PARK*, *OPL_FO_MLC*, *OPL_FO_RTB_HDG*, *OPL_FOLL_BANK*, *OPL_FOLL_HDG*
2	*OPL_FO_ATOL_TO*, *OPL_FO_ATOL_TOABRT*, *OPL_FO_ATOL_LAND*, *OPL_FO_ATOL_GOARND*
3	*OPL_FO_HLC*, *OPL_FO_RTB_GPS*, *OPL_FOLL_GPS*

4.5.2 Trajectory Control / Auto Flight Control System - Switch

The *Trajectory Control / Auto Flight Control System - Switch (TA-SW)* is controlling the input for the TC as well as the AFCS module. Its positions and flight control modes are listed in Table 4.18. The positions are used in the same way as TG-SW. The first position has a deactivating default input, while positions two and three are positions for ATOL- and SA-controlled modes respectively.

Table 4.18: *Trajectory Control / Auto Flight Control System - Switch*

Position	**TC / AFCS - Switch based on `mode_lgx`**
1	*STB*, *XBIT*, *OPL_FO_PARK*, *OPL_EP_GND*, *OPL_EP_AIR*, *OPL_EPLL_GND*
2	*OPL_FO_ATOL_TO*, *OPL_FO_ATOL_TOABRT*, *OPL_FO_ATOL_LAND*, *OPL_FO_ATOL_GOARND*
3	*OPL_FO_MLC*, *OPL_FO_HLC*, *OPL_FO_RTB_HDG*, *OPL_FO_RTB_GPS*, *OPL_FOLL_BANK*, *OPL_FOLL_HDG*, *OPL_FOLL_GPS*, *OPL_EPLL_AIR*

4.5.3 Inner Loop - Switch

As shown in Figure 4.20, the *Inner Loop - Switch (IL-SW)* is the most complex, with eight sources. An overview of its positions and flight control modes is given in Table 4.19.

The first position is used for operating modes that do not utilize the IL. Position six is used for all modes controlled by ATOL. The rest of the positions use various combinations of control allocations to enable the behavior described in Section 4.2. Those range from very restricted access in *OPL_FO_PARK* and *OPL_FOLL_GND* over partial direct-law in *OPL_EPLL_GND* to feed through from the TC and AFCS modules for the fully automatic modes covered by position three.

Table 4.19: *Inner Loop - Switch*

Position	**IL - Switch based on `mode_lgx`**
1	*STB*, *XBIT*
2	*OPL_FO_PARK*, *OPL_FOLL_GND*
3	*OPL_FO_MLC*, *OPL_FO_HLC*, *OPL_FO_RTB_GPS*, *OPL_FO_RTB_HDG*, *OPL_FOLL_GPS*, *OPL_FOLL_HDG*, *OPL_FOLL_BANK*
4	*OPL_EP_AIR*
5	*OPL_EP_GND*
6	*OPL_FO_ATOL_TO*, *OPL_FO_ATOL_TOABRT*, *OPL_FO_ATOL_LAND*, *OPL_FO_ATOL_GOARND*
7	*OPL_EPLL_GND*
8	*OPL_EPLL_AIR*

4.5.4 Actuator - Switch

The last injection switch, just before the OP is the *Actuator - Switch (AC-SW)*. It used source is, in contrast to all other switches, depending on `level1_lgx`. This simplifies the implementation, as shown in Table 4.20. In the first position, the default output is used, which disables the access for the flight control modules. In the second position, a special direct-law allocation or similar is used for defined experimental test cases. The third and default position is used for all operational modes.

Table 4.20: *Actuator - Switch*

Position	**AC - Switch based on `level1_lgx`**
1	*STB*
2	*XBIT*
3	*OPL*

4.6 Flight Tests

The FCSA is developed for use in both, UAVs and OPVs. Consequently, it is also tested on both types of aircraft. The UAV *SAGITTA* has an MTOM of $150kg$, a wingspan of $3m$, eight flight control surfaces, and two internal turbines. By contrast, the OPV *DA 42* has an MTOM of almost $2000kg$, a wingspan of about $14m$, and is a modified four-seater Part 23 Class II aircraft.

Before executing the first real-life flight tests, various testing and verification actions, as introduced in Section 3.4, were performed. Those tests included Unit Tests, Model Checking, Model in the Loop (MiL), Software in the Loop (SiL), Hardware in the Loop (HiL), and Aircraft in the Loop (AiL) simulations.

In the beginning, Unit Tests are used to define, simulate, and analyze test cases. For the FCSA, 13 Unit Tests were defined, which range from normal flight segments over takeoff abort scenarios to link loss simulations. Those have been used to frequently test the functional interaction of all FCSA state machines, without other external modules. A complete list with their test number and name is available in Appendix E. This is followed by Model Checking, which is built on formal methods, to detect design errors, generate or compliment test cases, analyze the model coverage, and prove key properties of the automation. The Simulink Design Verifier (SDV) is utilized in a bottom-up approach to guarantee continuous testing throughout the model. Additionally, the Modified Condition / Decision Coverage (MC/DC) coverage of all FCSA models is analyzed. Although it was not required by any third party, it was used to lower the possibility of errors during flight and to increase the confidence in the automation. This was especially important for *SAGITTA* since the first flight was already performed fully automatic.

Additionally, numerous MiL, SiL, HiL, and AiL simulations were performed and analyzed to eliminate even more errors. For *SAGITTA*, this was followed by multiple Ground Test campaigns, to increase the test scope as much as possible before the first flight. In the case of the *DA 42* partial functions of the FCS could be tested in flight, before moving to more critical flight phases like automatic takeoff and landing.

In the following, one flight test of each aircraft is presented as proof for the real-life applicability of the FCSA. The table of command history shows the sequential order of commands during the flight and the elapsed time. To guide comprehension, the name of the command and its new value is listed.

Furthermore, a numerical marker is listed in the table for a better allocation between the table and the figure. In this graphic, flight phases of automatic takeoff are drawn in red, while automatic landing ones are drawn in magenta. Additionally, the High-Level-Control mode is drawn in blue, while the Medium-Level-Control mode and superposition modes are drawn in cyan. Furthermore, ground tracks are drawn in green, the runway in yellow, and the projected flight trajectory to the ground in black.

In both depictions, the altitude is shown as Above Ground Level (AGL) and the x-axis is rotated to match the runway heading for better orientation.

4.6.1 SAGITTA

The validation of the intended full-scale flight mission was the goal of the *SAGITTA* project. The aircraft, introduced in Section 2.1, was therefore used for a real-life demonstration. Even though both, the FO and the EP, can control the aircraft, simulations had shown difficulties while manually controlling the aircraft. It was therefore decided to perform the maiden flight fully automatically. The EP was used as a backup in case of a major malfunction of the automatic system and to control the taxi of the aircraft.

After the design, implementation, and testing of the FCSA and other flight control modules, the UAV *SAGITTA* performed its maiden flight fully automatically on July 5, 2017. A picture of *SAGITTA* on the final approach of this flight is shown in Figure 4.21. Only a few days later the second flight from Runway 17 at Overberg Airport (FAOB) in South Africa, concluded the flight campaign. In both fully automatic flights, the FCSA administered the flight control modules and executed the commands of the EP and FO. The first flight consisted of a simple airfield traffic pattern but the mission already included over 40 commands from both pilots. The second flight extended this pattern and included an even greater variety of operational modes and commands to the FCSA.

An overview of the mode-related command history from both, the FO and the EP for the second flight is listed in Table 4.21. Even though the EP did not control the aircraft in the air, the taxi on the apron before and after the flight was performed manually. It can be seen, that over 60 commands from the pilots were used during this mission. Some of them are omitted from the list to increase the clarity. The resulting flight path, of this second and last flight of *SAGITTA*, is depicted in Figure 4.22.

Figure 4.21: *SAGITTA in Flight at Air Force Base Overberg (FAOB) in 2017 [Air2017]*

In the following, the history of commands and the resulting flight path are explained. The numerical markers in the command history, Table 4.21, and flight trajectory, Figure 4.22, are referenced in brackets in the text.

The EP was used in the beginning to taxi the aircraft from its startup location (1) to the takeoff position on the runway. Then the FO took over control. After performing some checks in parking mode, the takeoff was command by the FO (2). *SAGITTA* performed a fully automatic takeoff before the mode was changed to High-Level-Control (3). While in this mode various superpositioning modes of the FCSA were tested. At first, various altitude commands were used to alter the flight path height (3a-3b). Following those a loiter was initiated by the FO (3c), and performed by the aircraft. While in this loiter the command was retracted (3d) and the maneuver was finished as planned until reaching the entry point. After this circle, various speed superposition commands were tested by the FO (3e-3f). Then the link loss list was adjusted due to various constraints of the operational area (3g). When turning towards the runway, the automatic landing was activated (4) and the aircraft completed its second fully automatic mission. In the end, the mode Parking was activated (5), before the EP took over control again.

Table 4.21: *SAGITTA Second Flight - Command History*

Number	Time [s]	Command	Value	Marker
21	2940.767	`ep_inLoop_cmd_flg`	1	1
22	2940.837	`ep_inLoop_cmd_flg`	0	-
33	3364.393	`fo_inLoop_cmd_flg`	1	-
34	3365.279	`fo_inLoop_cmd_flg`	0	-
39	3968.883	`opl_mode_lgx`	40	2
40	4042.639	`opl_mode_lgx`	22	3
41	4120.818	`vertical_byFO_flg`	1	3a
46	4170.469	`vertical_byFO_flg`	0	3b
47	4177.496	`loiter_flg`	1	3c
48	4230.247	`loiter_flg`	0	3d
49	4286.748	`speed_byFO_flg`	1	3e
58	4317.205	`speed_byFO_flg`	0	3f
59	4322.217	`linkLossList_idx`	1	3g
60	4375.303	`opl_mode_lgx`	42	4
61	4491.694	`opl_mode_lgx`	20	5
62	4687.910	`ep_inLoop_cmd_flg`	1	-
63	4687.980	`ep_inLoop_cmd_flg`	0	-
65	5177.248	`fo_inLoop_cmd_flg`	0	-

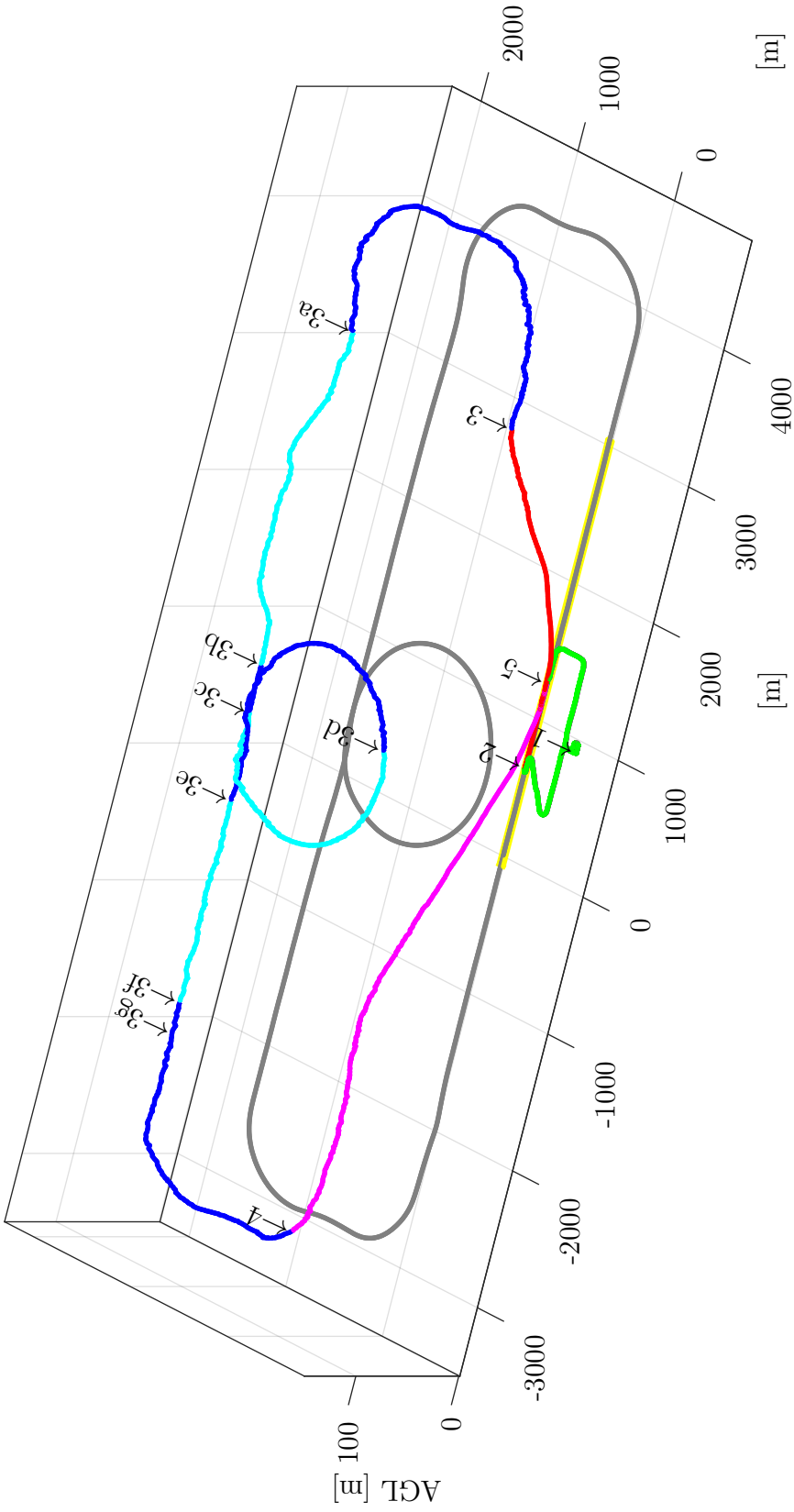


Figure 4.22: *SAGITTA Second Flight - Flight Trajectory*

4.6.2 DA 42

After its first automatic flight in January, 2016, the *DA 42*, introduced in Section 2.2, has been used in various projects at the Institute of Flight System Dynamics (FSD). In contrast to *SAGITTA*, the level of automation could be gradually increased, since pilots onboard the aircraft could be used for unsupported parts of the flight and for backup.

In the following, a flight test from January, 2017, will be used to prove the real-life applicability of the FCSA, within an OPV. In this flight, the EP has not been utilized and all phases are performed by the FO. However, over 30 commands have been used, some of which are only available within the OPV version of the FCSA. A picture of a final approach of the *DA 42* during such a flight test is shown in Figure 4.23. Additionally, the command history of the FO is listed in Table 4.22. Since only the FO is controlling the aircraft during this flight, initial control takeover signals and some others are omitted from the list. Furthermore, the trajectory of this flight test is depicted in Figure 4.24.

Within the OPV-enabled version of the FCSA for the *DA 42*, there is a special control mode for a pilot in the GCS with a control stick. Due to the modular implementation of the FCSA this additional feature and requirement could be implemented fairly easily. This mode uses a combination of bank-angle and load-factor command of the IL together with the ATHR of the AFCS to create a gamma-dot and chi-dot command control law that is usable in a higher latency scenario. In the following explanation, this is referred to as curvature-law. Since it uses the same control loops as the FO modes and to allow sharing of aircraft control, this is not activated by using the EP modes but rather by settings specific values for the `lateral_byFO_lgx` and `vertical_byFO_lgx` variables.

Figure 4.23: *DA 42 in Flight at Wiener Neustadt East Airport (LOAN) in 2017*

However, to make this possible at all, the limitation of values for those two control variables is extended by parameters, in the OPV version of the FCSA. This is one case in which the developed Intermittent Range Check (c.f. Subsubsection 3.2.4.2) is used to guarantee, that only valid modes, with respect to the project, are used.

At the beginning of the flight test, Parking is activated and followed by takeoff shortly afterward (1). After the fully automatic takeoff was performed the mode is switched to High-Level-Control (2), but quickly changed again to Medium-Level-Control (3), in which the FO is able to command altitude, heading, and speed. Following the climb and first turn in the direction of the testing area, various different commands are tested. Firstly heading command is changed to flight-track (3a) and then the altitude is changed to vertical-speed (3b). In the following the special command law for the EP, curvature-law, is tested (3c). In this case, the control stick provides flight path rate and curvature commands. This mode is deactivated when changing back to altitude- and track-based commands (3d). In the following, High-Level-Control mode is activated (4) and a loiter pattern is initialized (4a). When reverting this command (4b), while still being in the loiter patter, the aircraft finishes the current round and resumes to the next waypoint, when crossing the entry point. The aircraft is then guided back towards the airport using Medium-Level-Control (5), before activating automatic landing (6). After finishing this fully automatic mission, the FCSA is set to standby (7), which deactivates all flight control modules.

Table 4.22: *DA 42 Flight Test - Command History*

Number	Time [s]	Command	Value	Marker
08	1306.351	`opl_mode_lgx`	20	-
09	1317.991	`opl_mode_lgx`	40	1
12	1380.469	`opl_mode_lgx`	22	2
15	1402.870	`opl_mode_lgx`	21	3
17	1656.550	`lateral_byFO_lgx`	3	3a
18	2162.887	`vertical_byFO_lgx`	2	3b
20	2307.847	`vertical_byFO_lgx`	20	3c
21	2307.847	`lateral_byFO_lgx`	20	-
24	2460.447	`lateral_byFO_lgx`	3	3d
25	2461.027	`vertical_byFO_lgx`	4	-
26	2488.387	`opl_mode_lgx`	22	4
28	2789.115	`loiter_flg`	1	4a
29	2930.967	`loiter_flg`	0	4b
31	3742.505	`opl_mode_lgx`	21	5
33	3868.483	`opl_mode_lgx`	42	6
36	4059.804	`opl_mode_lgx`	1	7

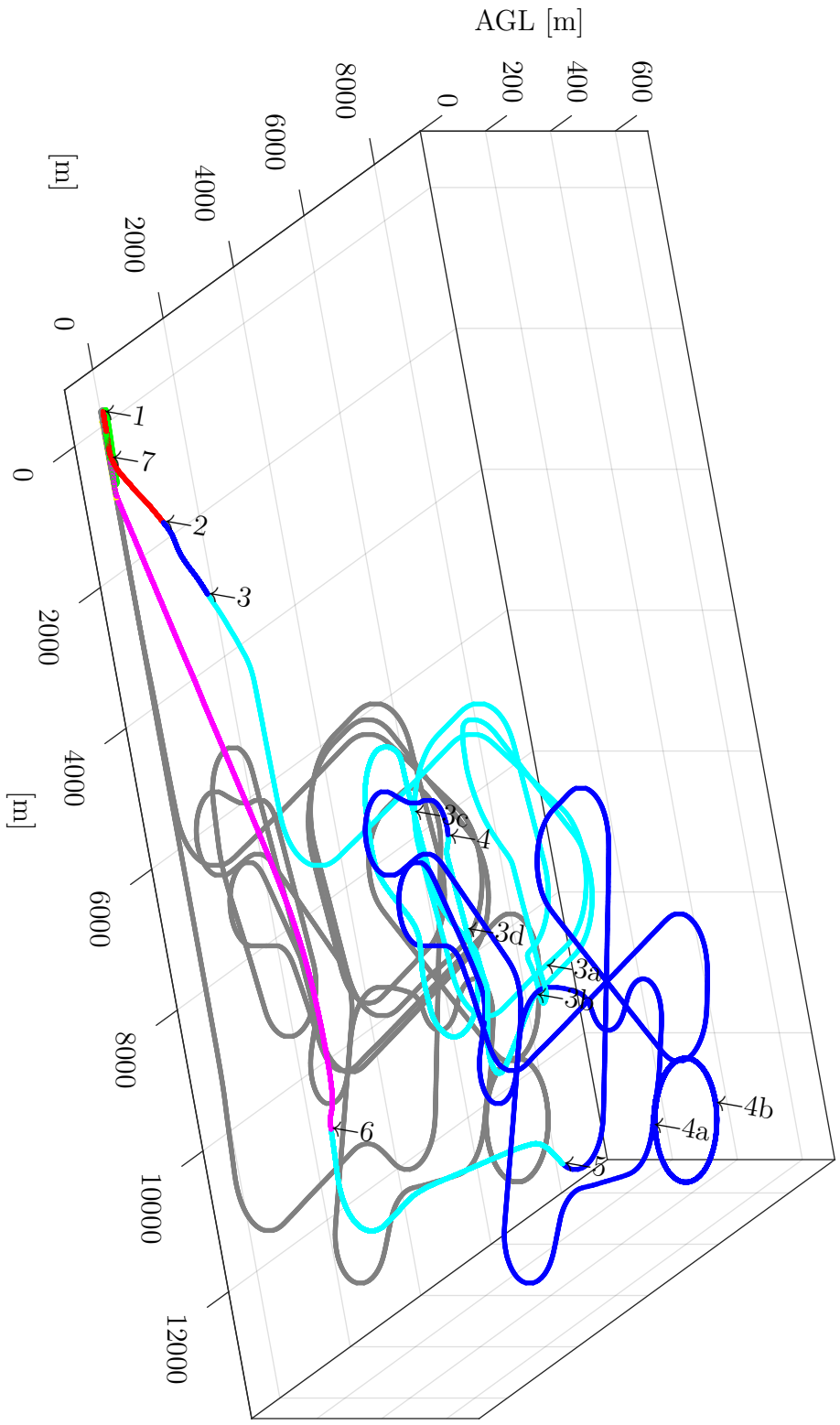


Figure 4.24: *DA 42 Flight Test - Flight Trajectory*

4.7 Summary

This chapter presents the *Operator-Centric Multi-User Flight Control System Automation for experimental UAVs and OPVs with Contingency Procedures (FCSA).*

In the beginning, the system architecture with respect to hardware, the Flight Control Computer (FCC), and the modules themselves is presented. This includes both aerial testbeds, *SAGITTA* and the *DA 42*, and the introduction of each flight control loop. These include *Input Processing and Monitoring (IPM)*, the *System Automation (SA)*, the *Automatic Takeoff and Landing (ATOL)*, the *Trajectory Generation (TG)* and *Trajectory Control (TC)*, the *Auto Flight Control System (AFCS)* including *Autothrottle (ATHR)*, the *Inner Loop (IL)*, and *Output Processing (OP)* module. The integration of the FCSA within this cascaded control loop and the placement of the injection points is the first part of contribution *C2.1 - Strategy for switchability between various modes on different authority levels, enabling experimental automation.* Additionally, the module architecture describes the integration of the generic modules into project-specific frameworks.

This is followed by a detailed description of all operational modes and their transition conditions and actions. It consists of over 15 operational modes on four levels and over 50 transition conditions and actions, which enable the behavior of the FCSA. Important modes include the basic differentiation between standby and operational as well as ground and airborne modes for the pilot. Additionally, various operator modes from parking, over autopilot and waypoint-based flight modes, to fully automatic takeoff and landing modes are discussed. This constitutes the contribution *C2.2 - Operational management concept for multi-user experimental OPVs and UAVs, increasing mode awareness.*

Those sections also include automatic contingency modes, which are divided into operational and malfunction contingency procedures. This allows for continued automatic flight and reduced workload for the pilot or operator. In the next section, the *Loiter Automation (LA)*, which is an encapsulated contingency module is introduced. Together with the other contingency procedures and modes, it reflects contribution *C2.3 - Automatic operational and malfunction contingency procedures for continuous operation in non-nominal circumstances.*

In the following, the injection switches are presented, which enable the injection and routing of commands within the cascaded control loop. They are controlled by the FCSA and placed between each adjacent control loop, to allow for access to all levels and complete contribution *C2.1.*

The chapter concludes with real-life flight test data from both demonstration platforms. The Unmanned Aerial Vehicle (UAV) *SAGITTA* performed its maiden flight fully automatic from takeoff to touchdown with the help of the FCSA. Furthermore, flight test data from the Optionally-Piloted Vehicle (OPV) *DA 42* is presented. Both prove the real-life applicability of the methodology, presented in Chapter 3, and the FCSA, which was used during countless flight tests of both demonstration platforms to administer the flight control modules and perform various missions.

Chapter 5

Flight Test Maneuver Injection

The developed *Multi-Maneuver Multi-Control-Level Flight Test Maneuver Injection with automatic Trim Point Capture and Verification (FTMI)* is presented in this chapter.

A comparison between a test pilot and the FTMI is shown in Figure 5.1. The specified maneuver, for both real-life flight tests, is an elevator doublet with an amplitude of 0.5°, a step time of $5s$, and a hold time of $20s$. While the maneuver can hardly be identified, when reviewing the data from the test pilot, the specified signal is clearly visible, when being executed automatically by the FTMI. The remaining offset and high-frequency noise are founded in low actuator and sensor performance as well as inadequate linking between the actuator and surface.

The author would like to thank *Christoph Göttlicher* for the collaborative work during the development of the FTMI. With his development of the maneuvers and the discussions, he has made a large contribution to the success of the FTMI.

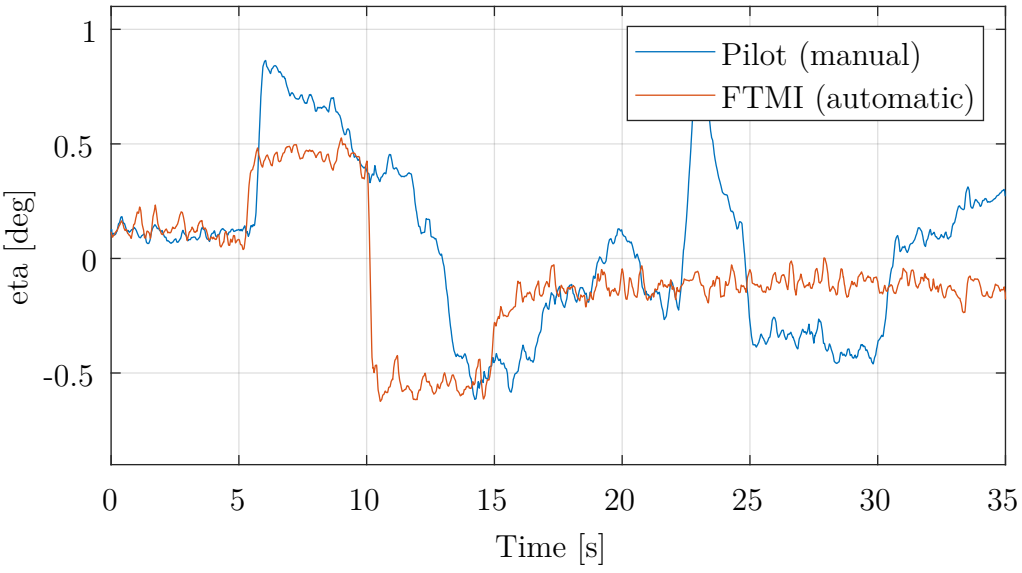


Figure 5.1: *Elevator Doublet Comparison*

Motivation - Flight Test Maneuver Injection

- Automatic flight test generation is highly favorable for simple maneuvers and inevitable for more complex maneuvers
- Numerous highly customizable maneuvers and various injection points are not supported by current solutions
- Recent developments do not incorporate advanced functions for safe and efficient flight testing

The current state of the art with respect to the flight test maneuver injection part of this thesis is introduced in Subsection 1.3.3. The resulting motivation, due to the identified deficits, is recapitulated here. It is related to the favorability of automated flight tests, customizable maneuvers, and advanced functions. Due to the increased precision, automated flight test generation is highly favorable even for simple maneuvers. For complex command patterns in flight tests, automation is inevitable due to the lack of human controllability. Additionally, current solutions don't offer multiple highly customizable maneuvers and various injection points throughout a cascaded control loop. Furthermore, advanced functions are not incorporated currently and can be integrated to increase flight test efficiency.

The derived objectives, from the deficits in the current state of the art, are presented in Subsection 1.4.3 and summarized in the following. The main objective of the FTMI is to provide automatic execution of flight test maneuvers within the Flight Control Computer (FCC). The software module needs to support complex-signal maneuvers and accurate execution of those. To increase the scope of application, the integration of numerous maneuvers and various injection points within the cascaded control loop is necessary. The FTMI is used in various short-term development projects and therefore shall support advanced features for efficient flight testing with increased safety to support rapid development and enable in-flight verification of Flight Dynamic Models (FDMs) and controller performance.

Objectives - Flight Test Maneuver Injection

- Provide an automatic and accurate execution of complex-signal flight test maneuvers and the integration within the FCC
- Support the integration of numerous maneuvers and various injection points within the cascaded control loop
- Develop advanced features for safe and efficient flight testing, like individual trim points for each maneuver

Contributions - Flight Test Maneuver Injection

- C3.1 - Generic design pattern, providing a free maneuver parametrization without reimplementation
- C3.2 - Dynamic flexible choice of injection points on multiple control levels, enabling a generic implementation and safe execution
- C3.3 - Individual trim point verification and automatic trim point capture for safe and effective flight testing

The contributions of the maneuver injection, presented in this chapter, are introduced in Subsection 1.5.3. The assignment of the individual contributions to the sections of this chapter is presented in the following together with the general outline.

At the beginning of this chapter, Section 5.1 describes the hardware architecture of both demonstration platforms used and the FCC *System Architecture* in combination with the integration of the FTMI in the cascaded control loop. Additionally, the software module architecture, which is associated with contribution *C3.1*, is presented.

In the following, the dynamic *Allocations Matrices and Injection Points*, which represent contribution *C3.2*, are described in Section 5.2. They enable a generic one-time implementation of the maneuvers in a centralized location, to allow for easy extendability and manageability among multiple developers as well as simplified testing.

Contribution *C3.3* is associated with the description of the *Operation Modes* and their *Transition Conditions and Actions*, which are presented in Section 5.3 and Section 5.4. The state machine-based approach enables the use of advanced features in certain modes, which are based on trim points for each maneuver. These include individual trim point verification for each maneuver and the possibility to delay the execution until this point is reached. Additionally, they utilize the outer control loops to provide automatic trim point capture for increased safety and efficiency.

The five currently implemented *Maneuvers* are Multi-Step, Multi-Ramp, Multi-Sine, Sweep, and Spline. Those maneuvers, which are used during the flight tests of the demonstration aircraft, are presented in Section 5.5.

Hereafter, example data from multiple *Flight Tests* of both demonstration platforms, *ELIAS* and the *Do 228*, is presented to prove the real-life applicability of the software.

The chapter is concluded with the *Summary* in Section 5.7, of the archived contributions and resulting applications of the FTMI.

The basic idea for this FTMI and parts of it have been previously published [KGH2018]. However, this chapter takes a more detailed approach and includes various undisclosed aspects of the maneuver injection.

5.1 System Architecture

This section introduces the overall system architecture of the *Multi-Maneuver Multi-Control-Level Flight Test Maneuver Injection with automatic Trim Point Capture and Verification (FTMI)*, which includes the platform-specific hardware architecture, the Flight Control Computer (FCC) system architecture as well as the software module architecture.

One main requirement is the platform-independent design and transferable implementation, without changing the internal structure, so as not to affect previous testing and verification. Therefore, the FTMI is tested on an ultralight Optionally-Piloted Vehicle (OPV) and a 19-seat twin-engine manned aircraft as introduced in Chapter 2. The hardware architecture of both is presented in Subsection 5.1.1.

In the following, the generic FCC system architecture is introduced in Subsection 5.1.2. This includes an overview of the relevant flight control modules and the integration of the FTMI as well as the dynamic allocation matrices and injection points within the cascaded control loop.

In Subsection 5.1.3 the software architecture of the FTMI is presented. This includes the modular design of the functional modules, used parameter structures for input and output, as well as the hardware requirement-based separation into two computational entities. This represents contribution *C3.1 Generic design pattern, providing a free maneuver parametrization without reimplementation.*

5.1.1 Hardware Architecture

The hardware architecture, in this part, refers to the FCC and the hardware connected to it. These can be sensors to get information about the aircraft's state, data links to receive data from and sent data to the Ground Control Station (GCS), as well as actuators to control the deflection of the surfaces of the aircraft.

Depending on the aircraft and project, different configurations and combinations are used. The FTMI is designed to be applicable to different types of aircraft and to support quick integration. Consequently, it is tested on two completely different aircraft, which are introduced in Chapter 2.

On the one hand, *ELIAS* is an ultralight OPV, which is treated according to national rules. As introduced in Section 2.3, the aircraft is a modified electric aircraft with a wingspan of $11m$. Its hardware architecture is described in Subsubsection 5.1.1.1.

On the other hand, the *Do 228* is a modified 19-seat Part 23 Class IV aircraft. This twin-engine commuter category aircraft with a maximum take-off mass (MTOM) of about $6000kg$ is introduced in Section 2.4. Its hardware architecture is described in Subsubsection 5.1.1.2.

5.1.1.1 ELIAS Architecture

ELIAS, introduced in Section 2.3, is used as an OPV demonstration platform for the FTMI. Figure 5.2 depicts a simplified overview of its hardware architecture. The overall hardware architecture is more complex and incorporates more devices and connections. However, this figure shows all relevant devices and connections for the FCC in the context of the FTMI in this thesis, which are sensors on the left, available data link connections on the top, and actuators and other mainly outgoing connections on the right.

The FCC receives most of its sensor information from the Navigation System (NAV). This not only includes data from the Inertial Navigation System (INS), but also data from an Air Data Computer (ADC), Magnetometer (MAG), and the Global Navigation Satellite System (GNSS). Information about the height above ground level is provided by the Radar Altimeter (RADALT). Additionally, the surface deflections are measured by the Data Concentrator Unit (DCU) and also provided to the FCC.

The aircraft is connected to a GCS via the Flight Control Data Link (FCDL). It is used to sent commands to the aircraft, as well as for receiving status and sensor data for monitoring purposes in the GCS.

Commands from the FCC are primarily sent to the three Actuator Control Electronics (ACEs), which are connected via Friction Clutches (FCs) and Electromagnetic Clutches (EMCs) to the surfaces, the latter of which is controlled by the Actuator Clutch Box (ACB). Additionally, a connection to another ACE is established to control the thrust lever. The connection to the ACB and ACEs is bi-directional, which allows the FCC to receive important information about e.g. their position or current consumption. Furthermore, a connection to the Multi-Function Display (MFD) is used to show information to the pilot, but cannot be used as an input to the FCC.

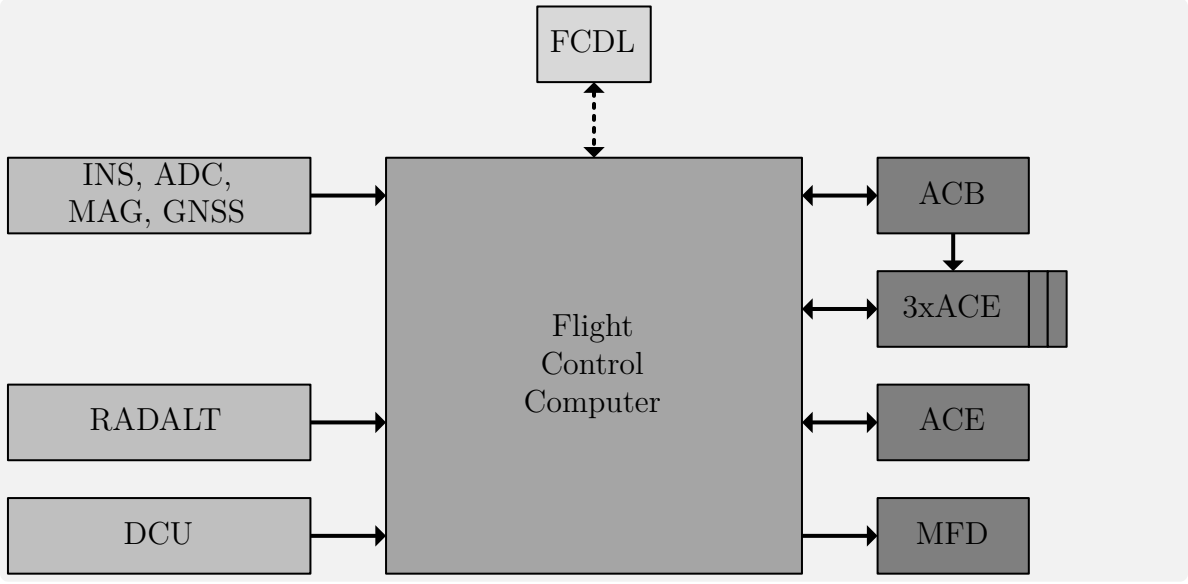


Figure 5.2: *ELIAS Hardware Architecture*

5.1.1.2 Do 228 Architecture

The *Do 228*, introduced in Section 2.4, is the only manned, in a classical sense, demonstration aircraft, referenced in this thesis. It is also equipped with the FTMI and therefore, in the following, more detailed information concerning the hardware architecture is presented. Figure 5.3 depicts a simplified overview as a basis for discussion in the following paragraphs and the rest of this chapter. The schematic uses the same distinction of inputs on the left, mainly outputs on the right, and bidirectional interfaces on the top.

As inputs, the *Do 228* uses a similar setup as the other demonstration platforms. The NAV includes an INS, a MAG, and a GNSS. It provides measurements of the aircraft's movement and orientation. Additionally, an ADC is used to gather data with respect to the surrounding air. Furthermore, a DCU is used to collect complementary data like control surface positions.

Since the *Do 228* is only used as a manned aircraft, not as an Unmanned Aerial Vehicle (UAV) or OPV, it is not equipped with any data link. However, a Mode Control Panel (MCP) is used to display information to the pilots and as an input interface for higher-level modes at the same time.

The calculated surface positions are sent to the three ACEs. They control the elevator, aileron, and rudder and are supervised by the Safety Relay Box (SRB). The SRB can also be directly controlled by the pilots, bypassing the FCC, to create an independent possibility to disconnect the EMCs and disengage the Flight Control System (FCS). Since the two throttle levers are not accessible by the FCC, the Thrust Director (TDR) is used to display the desired power setting to the pilots.

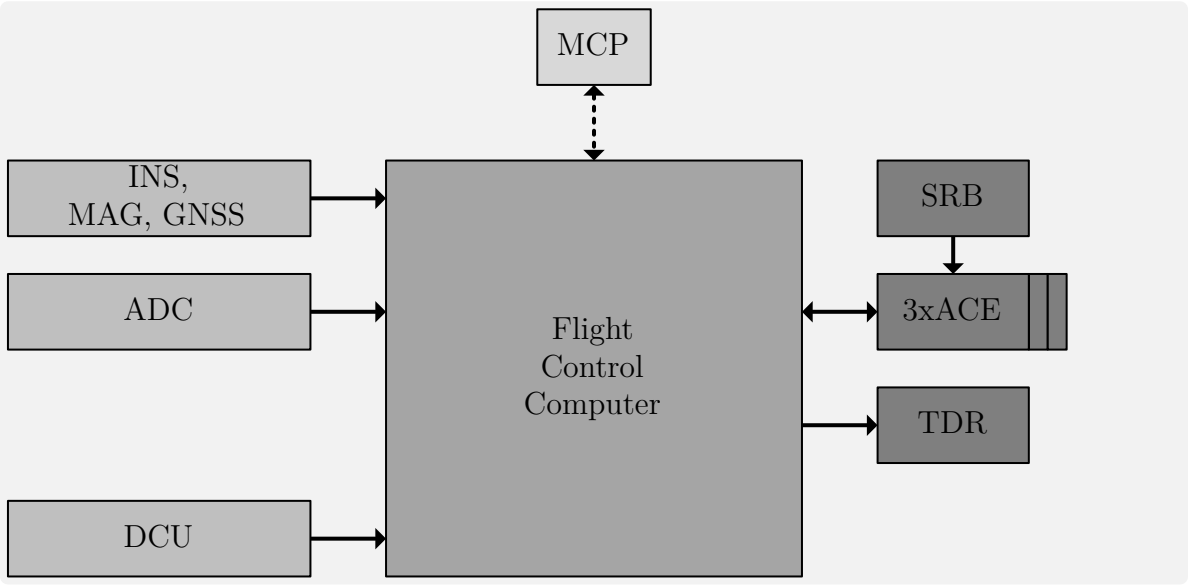


Figure 5.3: *Do 228 Hardware Architecture*

5.1.2 FCC System Architecture

In the following, the software architecture of the FCC and the integration of the FTMI are introduced. An overview is depicted in Figure 5.4. This includes the FTMI and its three injection modules: *Auto Flight Control System - Injection Module (AF-IM)*, *Inner Loop - Injection Module (IL-IM)*, and *Actuator - Injection Module (AC-IM)*. In contrast to Figure 4.5, fewer flight control loops are depicted since they are not relevant for the FTMI. Since the information about them can be found in Subsection 4.1.2, a further discussion is omitted here.

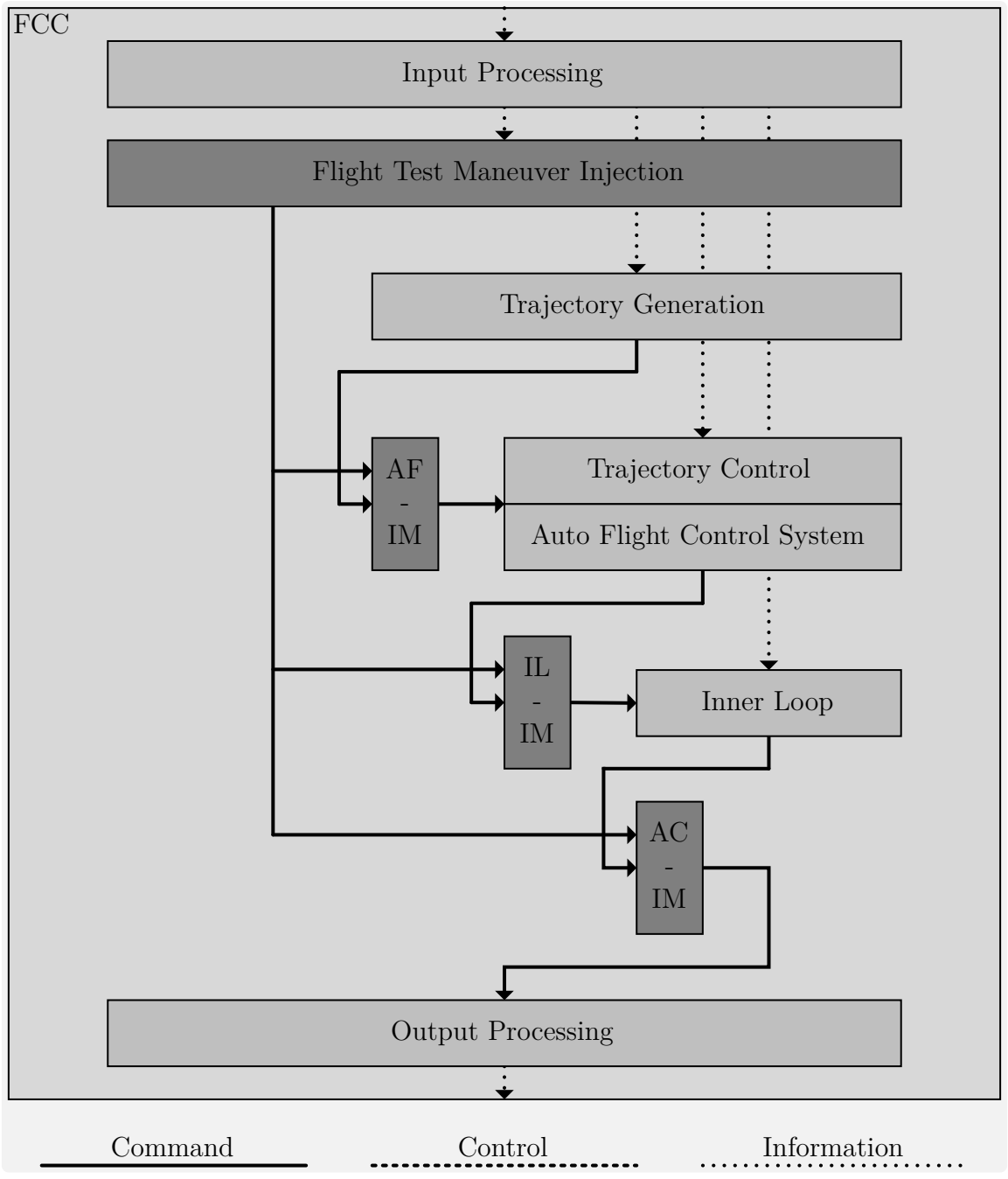


Figure 5.4: *FCC System Architecture*

5.1.3 Software Module Architecture

In the following, the software module architecture of the FTMI is described. The independent design pattern provides free maneuver parameterization without re-implementation and reflects contribution *C3.1 Generic design pattern, providing a free maneuver parametrization without reimplementation.*

An overview of this FTMI software module architecture is depicted in Figure 5.5. Besides the aircraft-specific *Input Processing* and *Output Processing*, it is split into two major sections. The *Control* section includes *Data Processing*, *Trim Point Calculation*, and the *State Machine* and is responsible for the overall control and planning of the maneuver, while the *Execution* section includes *State Execution*, *Maneuver Generation*, and *Protections* and is handling the real-time safety-critical execution.

This computational requirement-based separation is done to support future execution on two physically separated FCCs. The modules in the *Control* section need a larger amount of storage, due to the *Data Processing* module, which includes the saved maneuvers. However, this module does not need to be real-time capable. On the other hand, the modules in the *Execution* section need to be calculated in real-time but don't need a lot of storage. This is enabled by the developed asynchronous interface between the two, which only transfers data of the currently active maneuver and some additional status information. A detailed description of all modules is given in the following and the implementation of the *Control* and *Execution* module in *Simulink* and *Stateflow* is shown in Figure 5.6 and Figure 5.7.

The FTMI is designed to be used by a Flight Operator (FO) within a GCS. However, it can also be commanded by a flight test pilot or engineer onboard the aircraft. The following description within this chapter refers to the FO but can be replaced with alternative users.

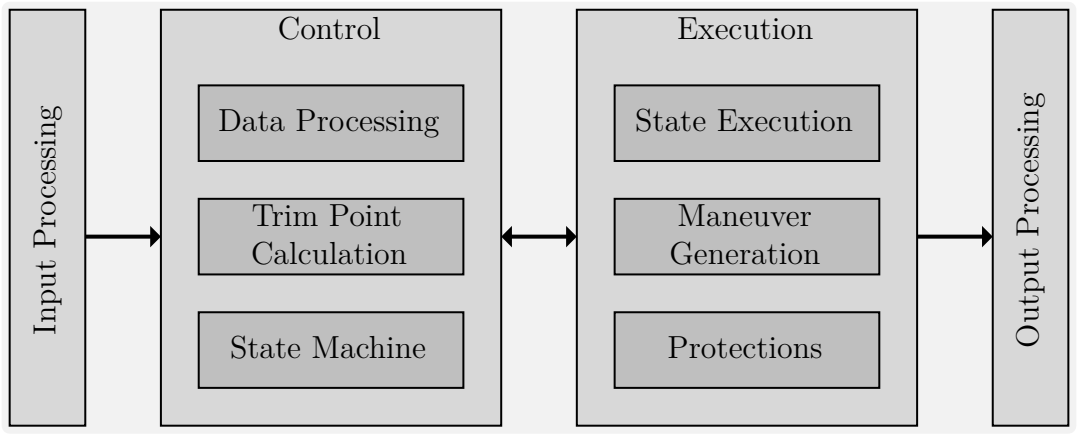


Figure 5.5: *Software Module Architecture*

5.1.3.1 Input Processing

The *Input Processing* module is used to convert aircraft-specific buses or structures to the generic bus structure of the FTMI. This includes the selection of relevant sensor information and the mapping of commands from the FO or onboard user. If certain functions shall be disabled for the respective platforms, parameters can be overwritten at this point as well. The *Input Processing* enables the usage of the FTMI in different projects without changing the internal structure.

5.1.3.2 Data Processing

The *Data Processing* module is part of the *Control* section and responsible for storing all maneuvers, selecting the currently necessary one, and processing and forwarding the data to the other modules within the *Control* section. It corresponds to the reddish part in Figure 5.6. Each maneuver consists of four fields, which structurally group the maneuver data. Those are `options`, `signalConfig`, `trimPoint`, and `data`.

The `options` part includes general information like the maneuver index, type of maneuver, and trim point configuration.

The `signalConfig` field includes the signal configuration and allocation data. Each maneuver consists of up to five signals, which can be mapped to arbitrary control surfaces. Additional data includes the configuration of the control surface, which includes the possible use of the currently active deflection as an offset. This can be used to keep the aircraft closer to the trim point and is explained in the relevant parts of Section 5.2.

The `trimPoint` field includes the target values for the trim point, e.g altitude, heading, speed, and more.

Specific parameters for the maneuver itself are located in the `data` field, which consists of 50 generic parameters, that have a different function for each maneuver type. This enables structured storage of the maneuvers, simpler implementation, and reflects part of contribution *C3.1*. After selecting the currently active maneuver, this data is mapped to *Simulink* buses and forwarded to the next modules.

5.1.3.3 Trim Point Calculation

The *Trim Point Calculation* module calculates if the individual trim point of the currently selected maneuver has been reached or captured. This is done by the greenish part of Figure 5.6. Each trim point consists of freely selectable values for altitude, heading, and speed of the aircraft, which can be activated individually. Additionally, a target value for the three turn rates of the aircraft can be selected. The latter ones are normally set to zero. If the error between the measured values of the sensors and the specified values of the trim point of the maneuver is below a predefined threshold for a certain amount of time, the trim point is considered reached. This result is forwarded to the *State Machine*.

5.1.3.4 State Machine

The *State Machine* is administering the FTMI and controlling the flight control loops during the capture of a trim point and execution of a maneuver. It is shown, with its surrounding parts, in the blueish area of Figure 5.6.

Six control modes are used to implement the required behavior, which are *Standby* (*STB*), *Wait-for-Trim-Point* (*WTP*), *Wait-for-Auto-Trim-Point* (*WATP*), *Execute-Maneuver* (*EMA*), *Wait* (*WAIT*), and *Index-Error* (*IDER*). Those six modes and their respective transition conditions and actions are described in more detail in Section 5.3 and Section 5.4.

5.1.3.5 State Execution

The *State Execution* is part of the *Execution* section. It is responsible for signal routing within the *Execution* section with respect to the currently active mode of the *State Machine*. This is done by the reddish part of Figure 5.7.

5.1.3.6 Maneuver Generation

The *Maneuver Generation* module is generating the flight test signals. It's implemented within the greenish area of Figure 5.7. Depending on the currently selected maneuver type the corresponding conditionally executed subsystem is activated. The author did not implement those maneuvers but provided the infrastructure to integrated numerous types of maneuvers without re-implementation.

5.1.3.7 Protections

Within the *Protections* module, different methods are implemented to keep the aircraft within the accepted flight envelope and as close to the trim point as possible. It's represented by the blueish part in Figure 5.7. Those protections include step commands, relative step commands, and low bandwidth control. Such protections need to be uncorrelated to the injected signal to retain the informational value of the maneuver. The development of suitable protections is not in the scope of this thesis and further information is, therefore, omitted here.

5.1.3.8 Output Processing

Similar to the *Input Processing*, the *Output Processing* is responsible for mapping internal status information to aircraft-specific interfaces and structures. Additionally, the command structures for the three injection modules, AF-IM, IL-IM, and AC-IM, are generated.

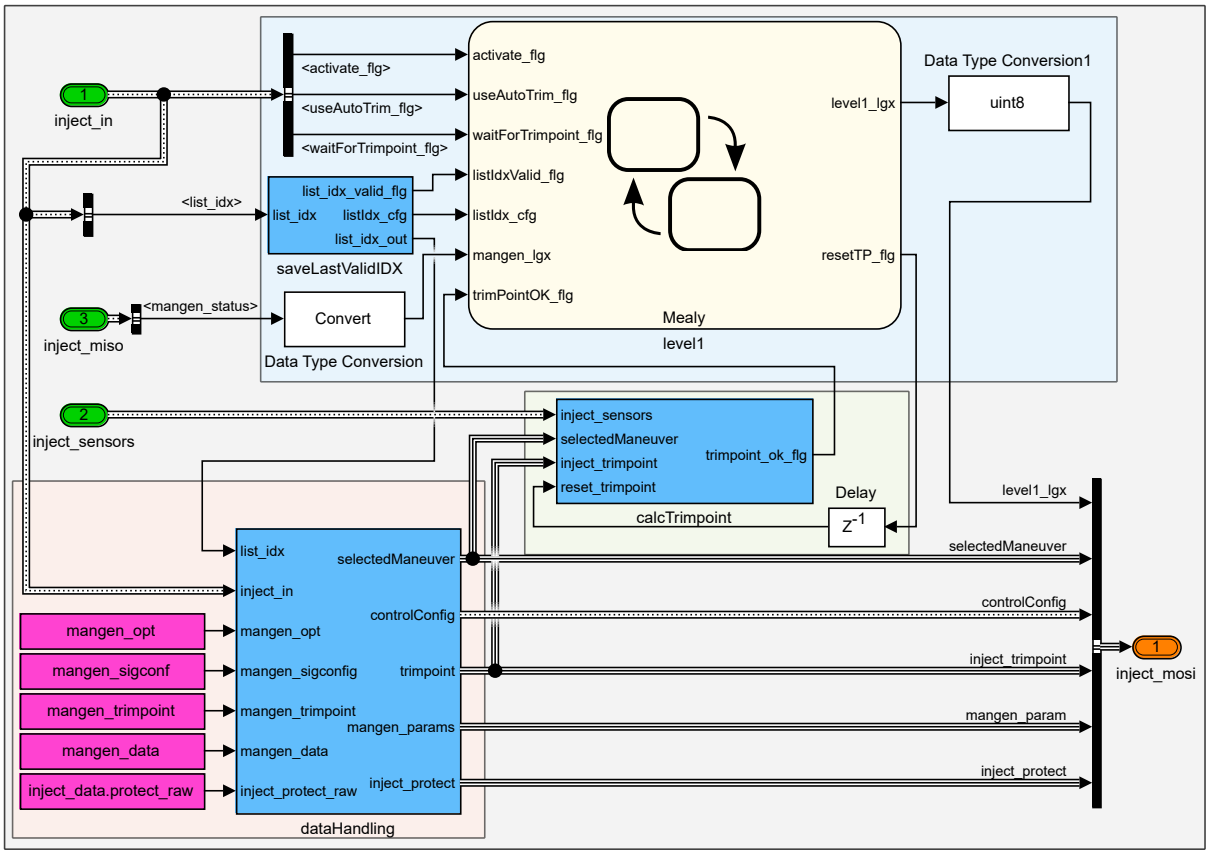


Figure 5.6: *Control Module*

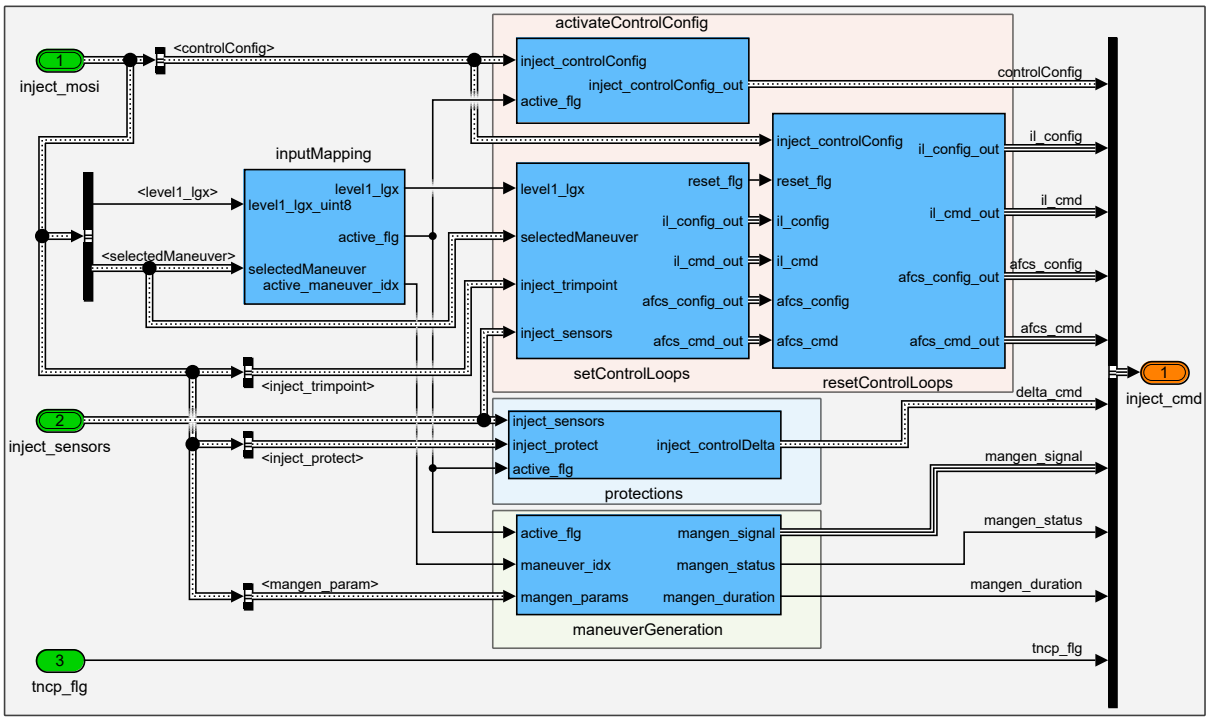


Figure 5.7: *Execution Module*

5.2 Allocation Matrices and Injection Points

This section takes a detailed look at the dynamic allocation matrices and injection points of the FTMI, which represent contribution *C3.2 Dynamic flexible choice of injection points on multiple control levels, enabling a generic implementation and safe execution.* The overall integration within the FCC has been introduced in Subsection 5.1.2. However, this section describes the specific implementation of all injection modules in more detail.

A schematic of the injection architecture, which is part of the FTMI, is depicted in Figure 5.8. In contrast to Figure 5.4, it can be seen that in general each Injection Module (IM) actually consists of an Override Switch (OR) and/or an Injection Switch (IJ). Even though not both of them are used on each level, the others could be added in the future.

The *Auto Flight Control System - Injection Module (AF-IM)* consists of the *Auto Flight Control System - Override Switch (AF-OR).* Therefore, the *Auto Flight Control System (AFCS)* can be utilized to perform advanced features like automatic trim point capture.

The *Inner Loop - Injection Module (IL-IM)* consists of the *Inner Loop - Override Switch (IL-OR)* and *Inner Loop - Injection Switch (IL-IJ).* Accordingly, the *Inner Loop (IL)* can be used for support functions but maneuvers can also be injected to test the control and tracking performance of the IL. Supporting functions can be used additionally to the maneuver to stabilize the aircraft in the not-excited axis.

The *Actuator - Injection Module (AC-IM)* consists only of the *Actuator - Injection Switch (AC-IJ)*, which allows for injection of maneuvers to the actuators of the aircraft.

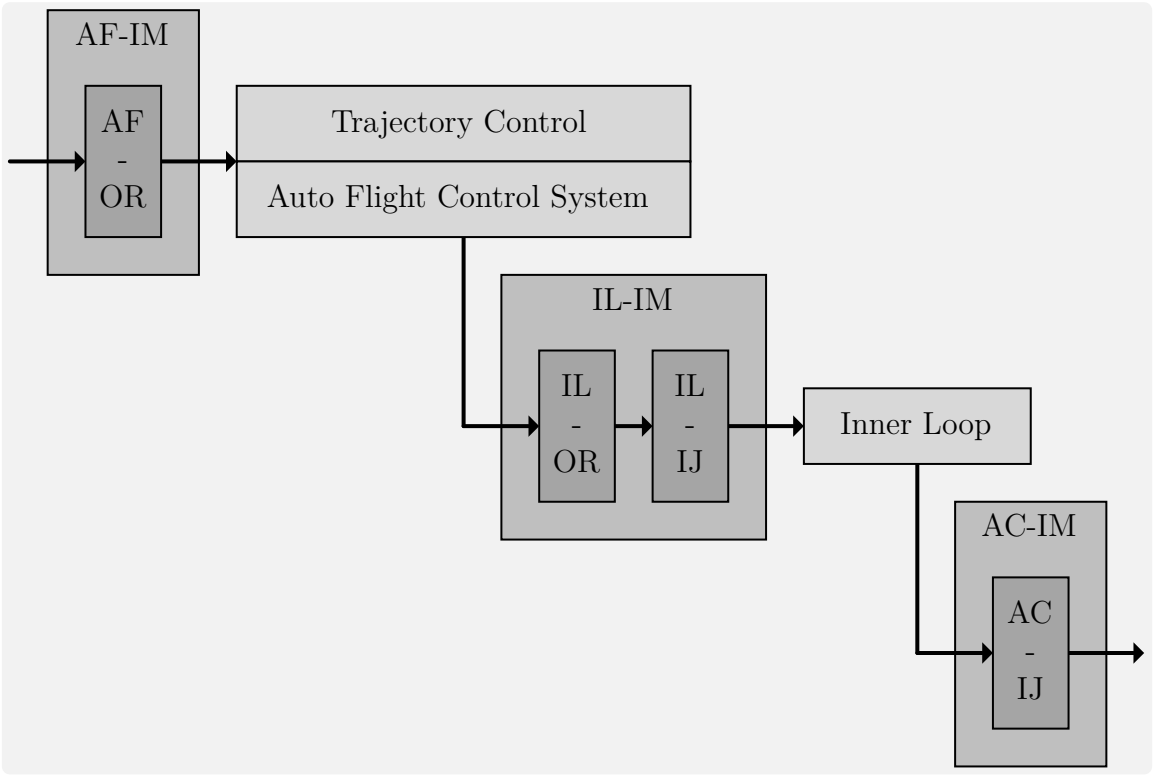


Figure 5.8: *Injection Architecture*

5.2.1 Auto Flight Control System - Override Switch

The AF-OR is used to feed alternative inputs to the AFCS. Those are used to enable the advanced functions of the FTMI, like automatic trim point capture. An overview of the implementation is depicted in Figure 5.9.

Relevant parts of the command interface to the AFCS, that can be changed by the FTMI, are selected in the Bus Assignment. The first three, `lnav_fms_cmd`, `vnav_fms_cmd`, and `engy_fms_cmd`, are enumerated mode selection commands. Those control the active mode in the respective axis. The following five variables, `ias_fms_cmd_mDs`, `psi_fms_cmd_rad`, `h_fms_cmd_m`, `phi_fms_cmd_rad` and `theta_fms_cmd_rad`, are the actual command values. Those are set to the specific trim point values if the AFCS is used for automatic trim point capture or to other predefined values for high-level support functions like altitude hold.

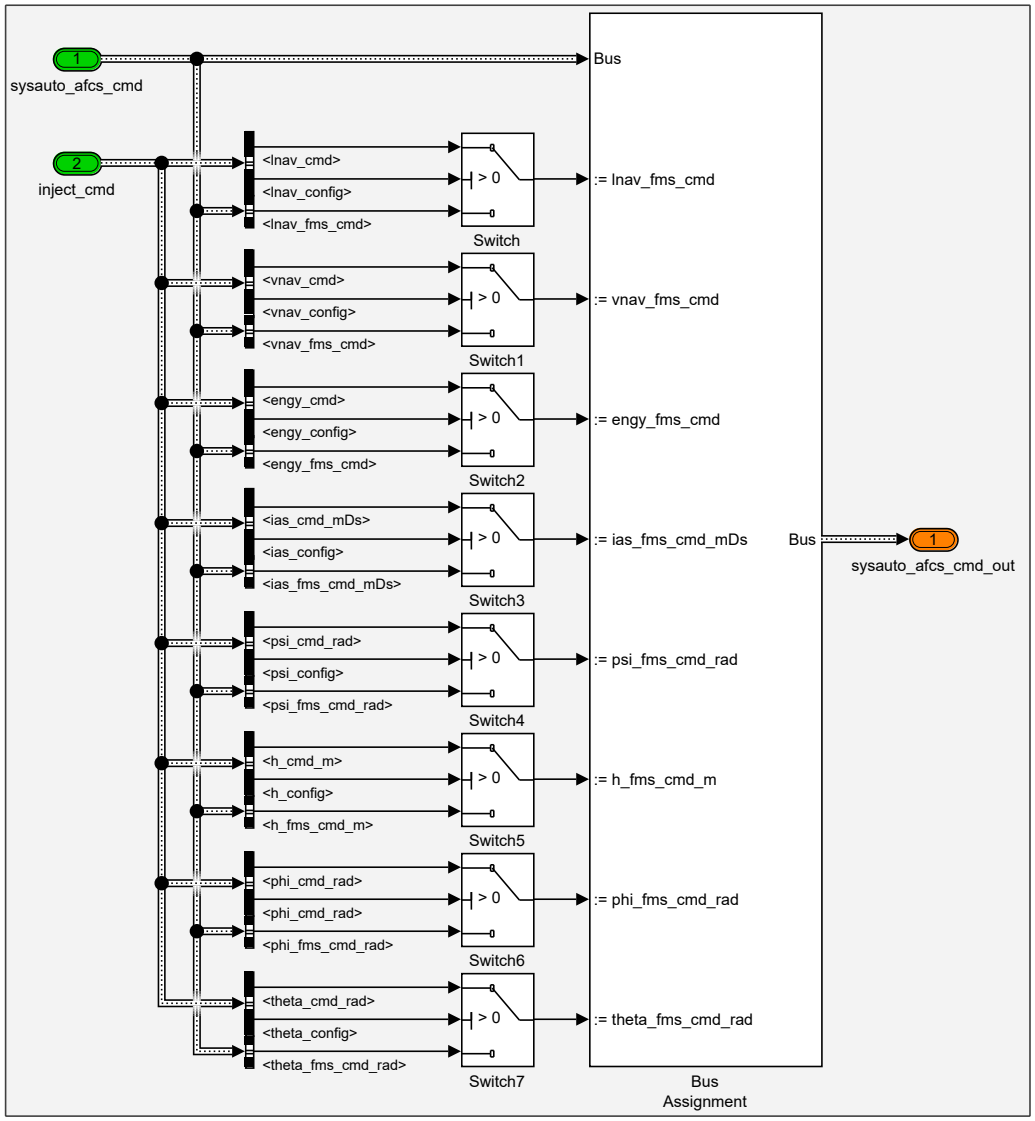


Figure 5.9: *Auto Flight Control System - Override Switch*

5.2.2 Inner Loop - Override Switch

The *Inner Loop - Override Switch (IL-OR)* is used if alternative commands need to be sent to the IL. Similar to AF-OR, the first three variables in the top of Figure 5.10, `il_yaw_axis_engaged_flg`, `il_roll_axis_engaged_flg`, and `il_pitch_axis_engaged_flg` are used to activate or deactivate the respective axis. The last three, `phi_cmd_rad`, `fDg_z_R_B_cmd`, and `fDg_y_R_B_cmd` are used to forward the specific command value to the IL.

Those variables are also used to realize low-level support functions. They include but are not limited to a wing-leveler, which can be used during vertical maneuvers, and a constant load-factor, which can be used during lateral maneuvers.

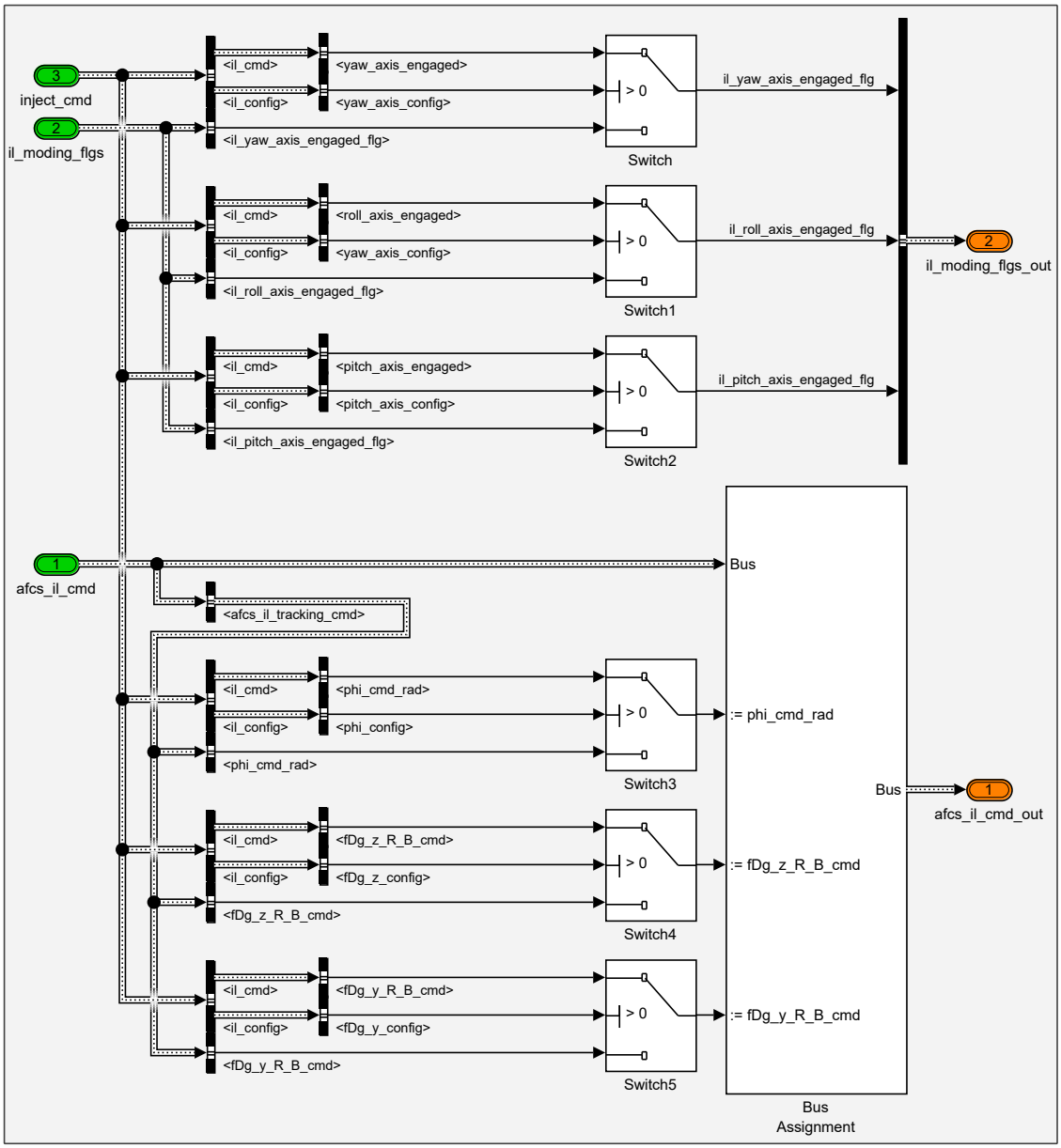


Figure 5.10: *Inner Loop - Override Switch*

5.2.3 Inner Loop - Injection Switch

The *Inner Loop - Injection Switch (IL-IJ)* is used to inject the generated maneuvers into the IL. Use cases could be performance testing or tracking analysis. An overview of the IL-IJ is depicted in Figure 5.12.

In the greenish area, it can be seen, that injection is possible to all three command values. Those are `phi_cmd_rad`, `fDg_z_R_B_cmd` and `fDg_y_R_B_cmd`. Each axis uses a similar structure with library blocks for the signal merging.

An exception is the generation of the `default_value` for `fDg_z_R_B_cmd`. Within this subsystem, located in the reddish area of Figure 5.12, either the externally provided value is selected or an internally calculated load-factor with turn compensation is forwarded. Such a calculation is only necessary for the load-factor because its default value is the only non-zero one.

This subsystem is shown in Figure 5.11. In the reddish part, the compensated trim load-factor is calculated. If both the pitch-angle (`theta`) and the roll-angle (`phi`) are zero, the necessary load-factor would be one or more correctly negative one. However, even in a straight and level flight, theta is most likely not zero, which results in a change of the load-factor. Depending on the situation, this can be used as an offset to the maneuver, which is even more relevant in real-life flight tests, since disturbances will slightly change the pitch and roll-angle throughout any test.

The subsystems, which consist of the dynamic signal allocation use some generic inputs as well as axis-specific parts. Depending on the `active_flg`, which can be different for each axis but only active during the execution of a maneuver, a switch is used to select between the original command from the adjacent control loop and the generated maneuver. A detailed explanation of those subsystems is given in Subsection 5.2.4.

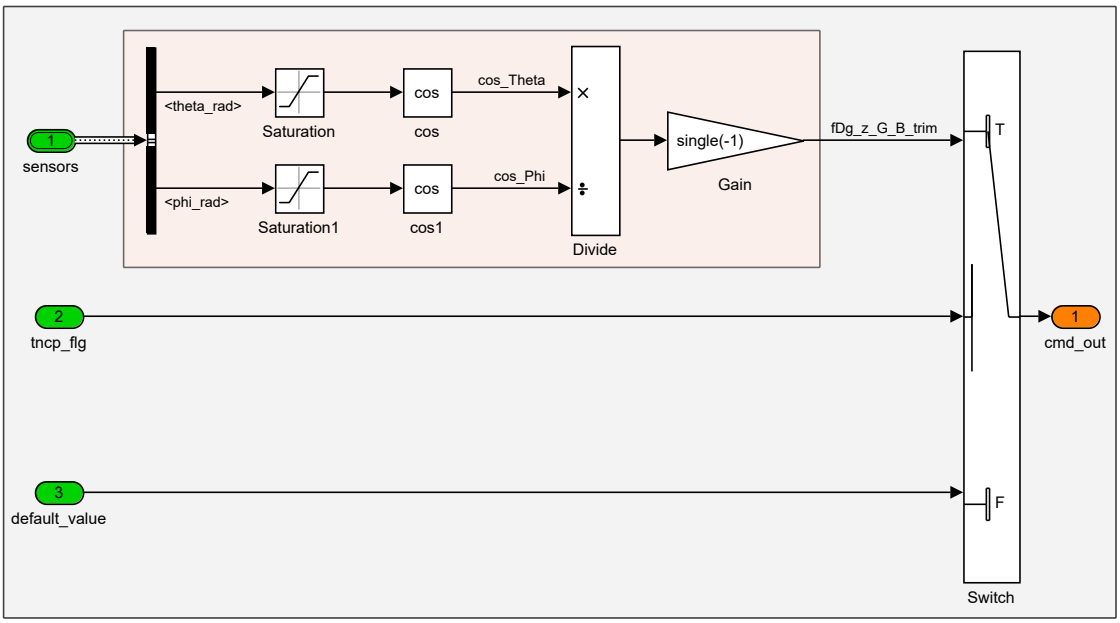


Figure 5.11: *Load-Factor Selection*

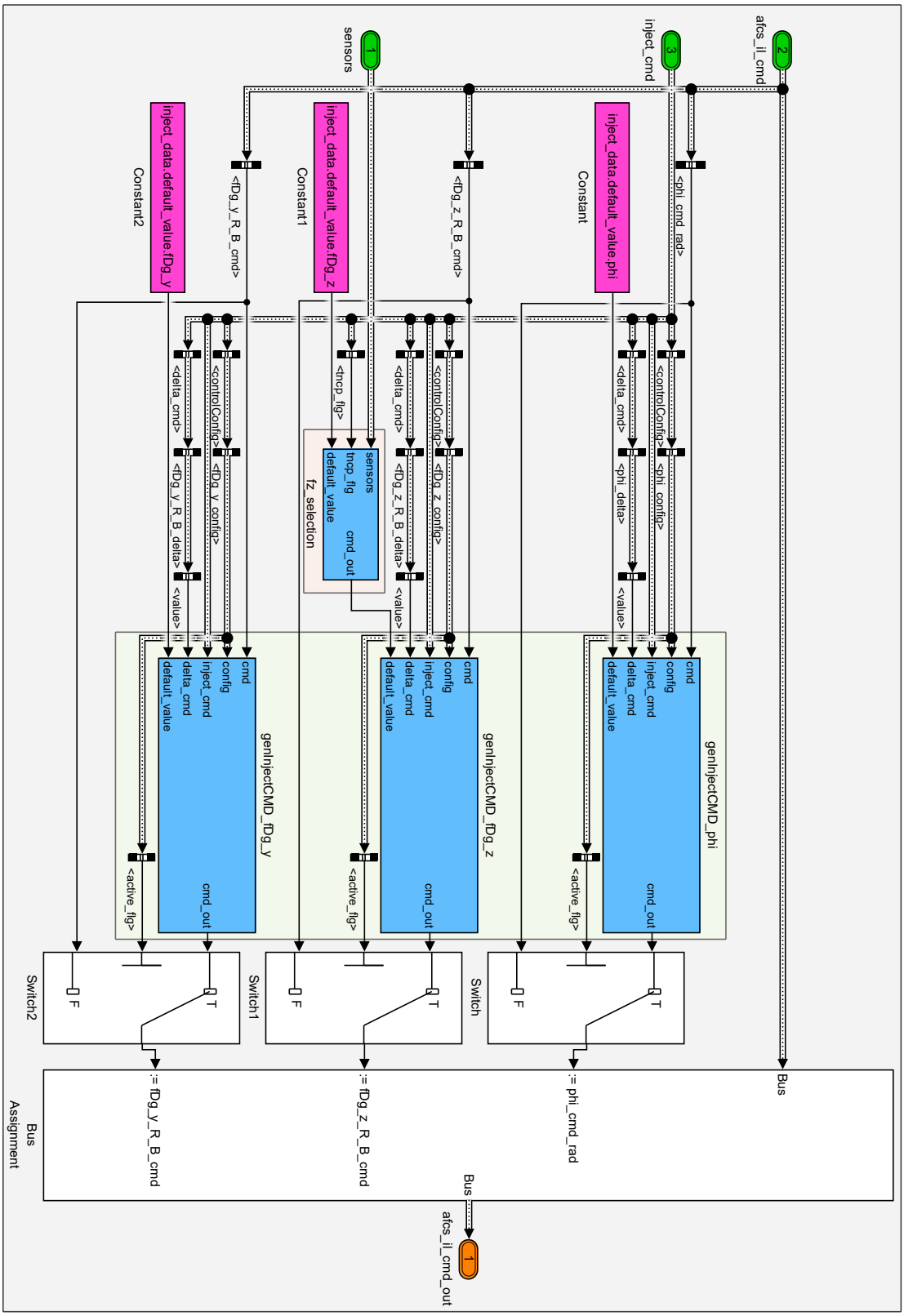


Figure 5.12: *Inner Loop - Injection Switch*

5.2.4 Actuator - Injection Switch

On the lowest level, the *Actuator - Injection Switch (AC-IJ)* is used to inject flight test maneuvers directly to the actuators. An overview is shown in Figure 5.14. Except for the fact, that other signals are used, the structure and implementation are similar to IL-IJ.

The merging and dynamic signal allocation subsystem, located in the greenish area of Figure 5.14, is shown in Figure 5.13. In the reddish part, the selection between a predefined default value and the trim value of the command is performed. If the current value of the command shall be used as an offset, its value at the beginning of the maneuver is saved within the "holdValue" block and added to the signal. Otherwise, a predefined default value (normally zero) is used, with the exception of the vertical load-factor as explained in Subsection 5.2.3.

In the greenish part, the relevant signal for the specific command is selected. Based on the individual configuration of the maneuver, one or more of the five source signals are selected to contribute to the output to the surface, or control loop input.

In the blueish part, all signals are added together. Additionally, the command from the protection algorithm is added if activated. The saturation of the signal, in the end, is omitted here since it is performed further down in the control allocation or interface mapping part anyway.

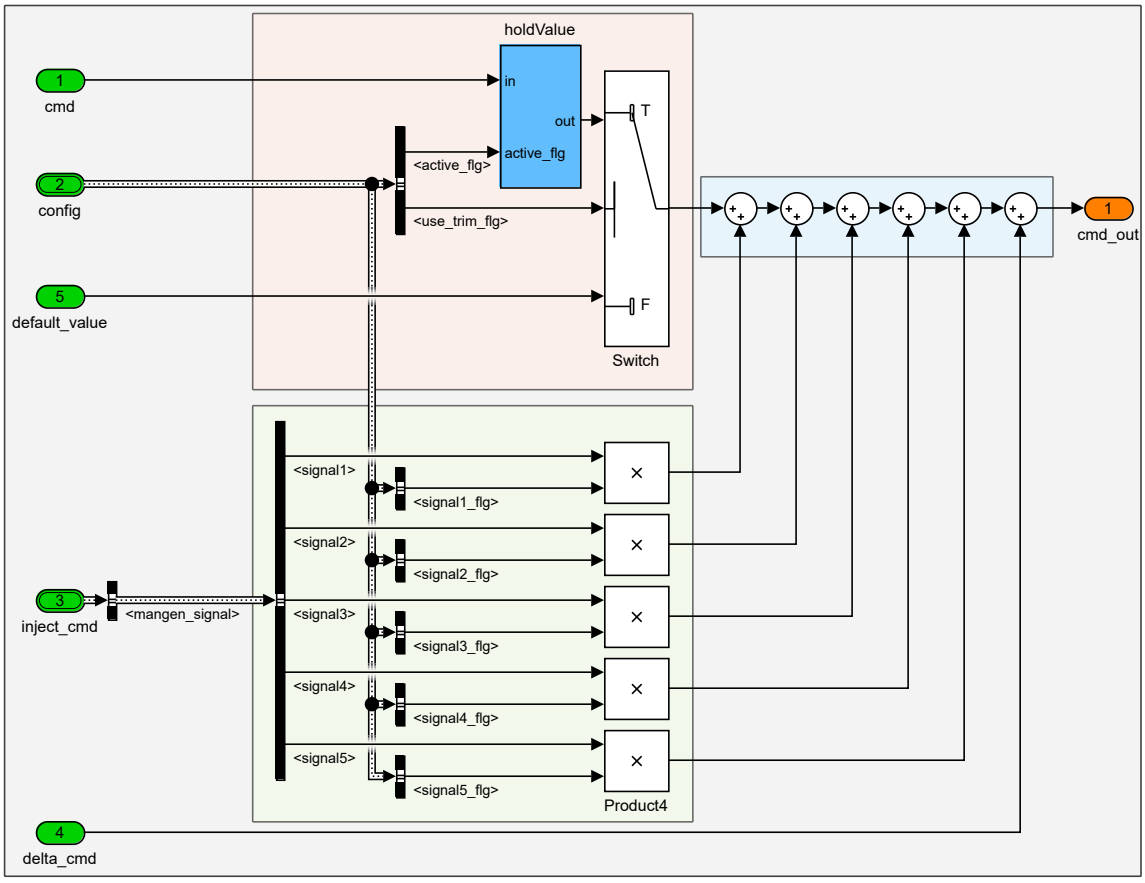


Figure 5.13: *Command Generation*

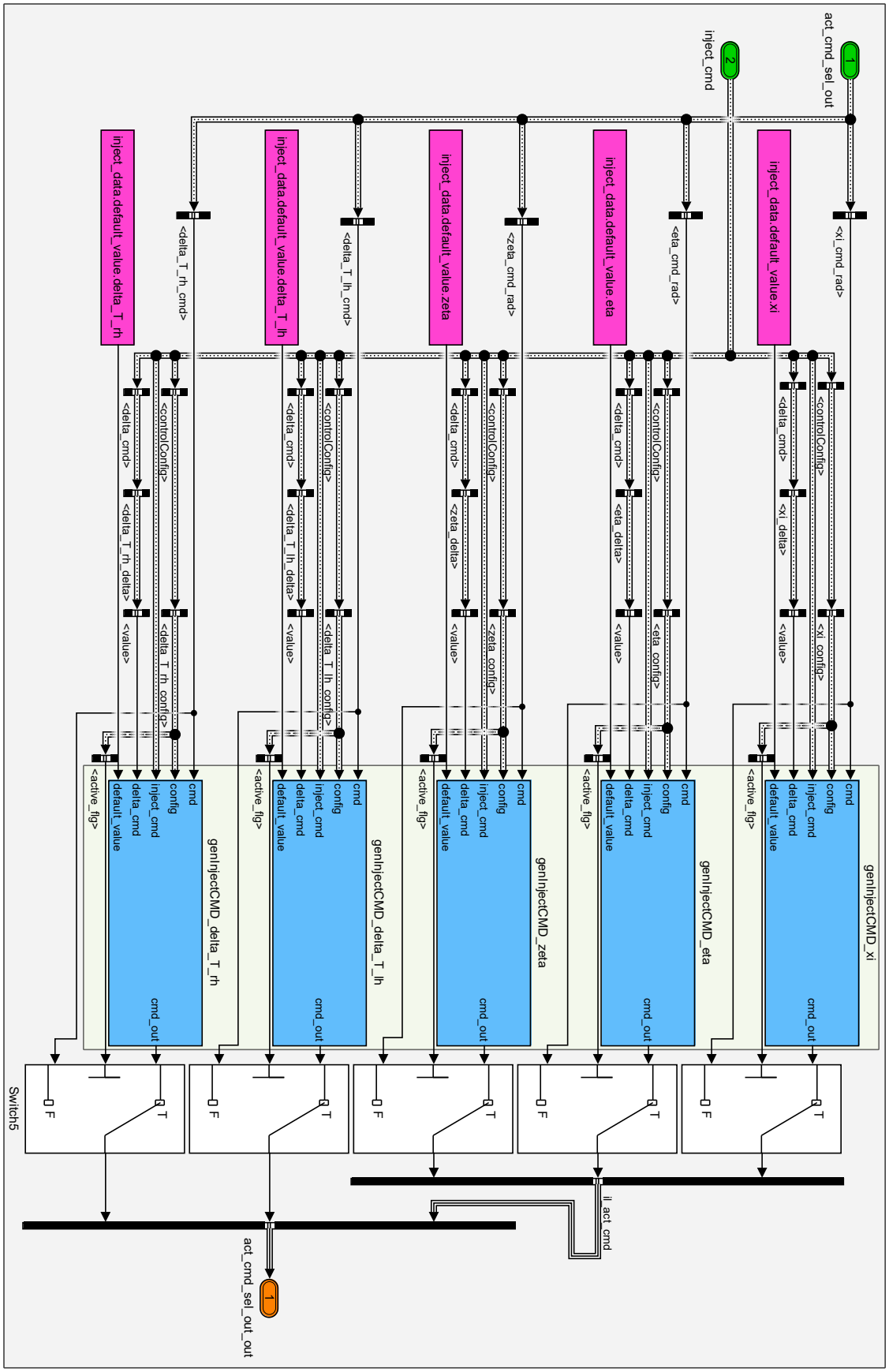


Figure 5.14: *Actuator - Injection Switch*

5.3 Operation Modes

The main state machine of the FTMI consists of six modes, which are used to realize the overall functionality of the developed software module. They are listed below, described in the following, and represent contribution *C3.3 Individual trim point verification and automatic trim point capture for safe and effective flight testing.*

- *Standby* (*STB*)
- *Index-Error* (*IDER*)
- *Wait-for-Trim-Point* (*WTP*)
- *Wait-for-Auto-Trim-Point* (*WATP*)
- *Execute-Maneuver* (*EMA*)
- *Wait* (*WAIT*)

The initial mode and the default mode of the FTMI, when not active, is *Standby* (*STB*). It can be active during both, manual flight and automatic flight.

Each maneuver has its own unique identifier, which is used during the definition of the maneuver as well as during the flight test to select the respective maneuver. If the selected index is not available the state machine will enter the *Index-Error* (*IDER*) mode.

Additional to the signal data, each maneuver also consists of an individual trim point, which includes a specific altitude, heading, and speed as well as maximum turn rates. If the maneuver has been activated, but the respective trim point has not been reached by the pilot, the FTMI remains in *Wait-for-Trim-Point* (*WTP*). The individual trim point elements can only be chosen during maneuver definition, the option to wait with the execution until the trim point is reached can be selected while airborne.

If the FTMI is used in a more advanced FCS, that already includes a working inner-loop and autopilot, the mode *Wait-for-Auto-Trim-Point* (*WATP*) can be used. During this mode, the necessary control loops are automatically activated and used to capture the trim point without any intervention by the pilot.

When the trim point is reached during *WTP*, *WATP*, or if the trim point was de-activated, the mode *Execute-Maneuver* (*EMA*) is activated and the maneuver itself is executed. The FTMI remains in this mode until the maneuver is complete.

If the maneuver was executed successfully or aborted due to an error, the mode *Wait* (*WAIT*) is entered. It is used to synchronize the FCC and the GCS. The maneuver execution is started using a boolean variable, which is normally controlled by a switch in the GCS. If the variable does not remain `true`, the maneuver is immediately aborted. Therefore, this switch will be active after the maneuver has been completed and can then be deactivated, using the mode information from the FTMI, thus synchronizing both devices, re-activating *STB*, and enabling the start of the next maneuver.

5.4 Transition Conditions and Actions

After previously describing the operation modes of the FTMI, this section focuses on the transition conditions and actions between those modes. It, therefore, represents the second part of contribution *C3.3 Individual trim point verification and automatic trim point capture for safe and effective flight testing.*

The *Stateflow* state chart, with the six previously introduced modes, *STB*, *IDER*, *WTP*, *WATP*, *EMA*, and *WAIT* and the possible transition paths between those six modes, is depicted in Figure 5.15. The necessary in- and outputs of this state machine, that are used by the transition conditions and actions, are listed in Table 5.1.

They include variables like the `activate_flg`, the `useAutoTrim_flg`, and the `waitforTrimpoint_flg`, which are sent from the FO in the GCS. They can be used online to modify the behavior of the FTMI depending on the next maneuver, its requirements, or the current situation. Additionally, an index is transmitted from the GCS, indicating the current maneuver. This is used to generate `listIdx_cfg` and `listIdxValid_flg`, which are used to determine a change and the validity of the index. Furthermore, the status from the maneuver generation (`manGen_lgx`) of the secondary module is used within the state machine as well as the evaluation of the current trim point (`trimPointOK_flg`).

The state machine generates two outputs. The currently active state is represented in `level1_lgx`, while `resetTP_flg` is used to reset the trim point evaluation. Furthermore, the transition matrix with all possible paths between the different modes is shown in Table 5.2. All transition conditions and actions are explained in the following. Their implementation is listed in Appendix F.

During the initialization of the state machine, the unconditional transition *init→stb* to *STB* is executed. This sets the respective `level1_lgx` and also sets `resetTP_flg` to `true`. In this state, the trim point evaluation is held in a reset state until needed.

From *STB* various other modes can be reached. For a simple activation of a maneuver without any trim point functionality, the transition *stb→ema* to *EMA* can be used. It is executed if the activation is requested, `activate_flg` is set to `true`, and `waitforTrimpoint_flg` is set to `false`.

Under normal circumstances, the activation of *EMA* triggers the execution of the selected maneuver. The end of the maneuver is indicated from the *Maneuver Generation* to the *State Machine* module with the respective value in `manGen_lgx`. This triggers the transition *ema→wait* and activates *WAIT*. However, this transition is also executed if the index for the flight maneuver is changed during execution. This would be recognized via `listIdx_cfg` and also activates *ema→wait*. The transition *ema→wait* sets `level1_lgx` to the respective value when entering *WAIT*.

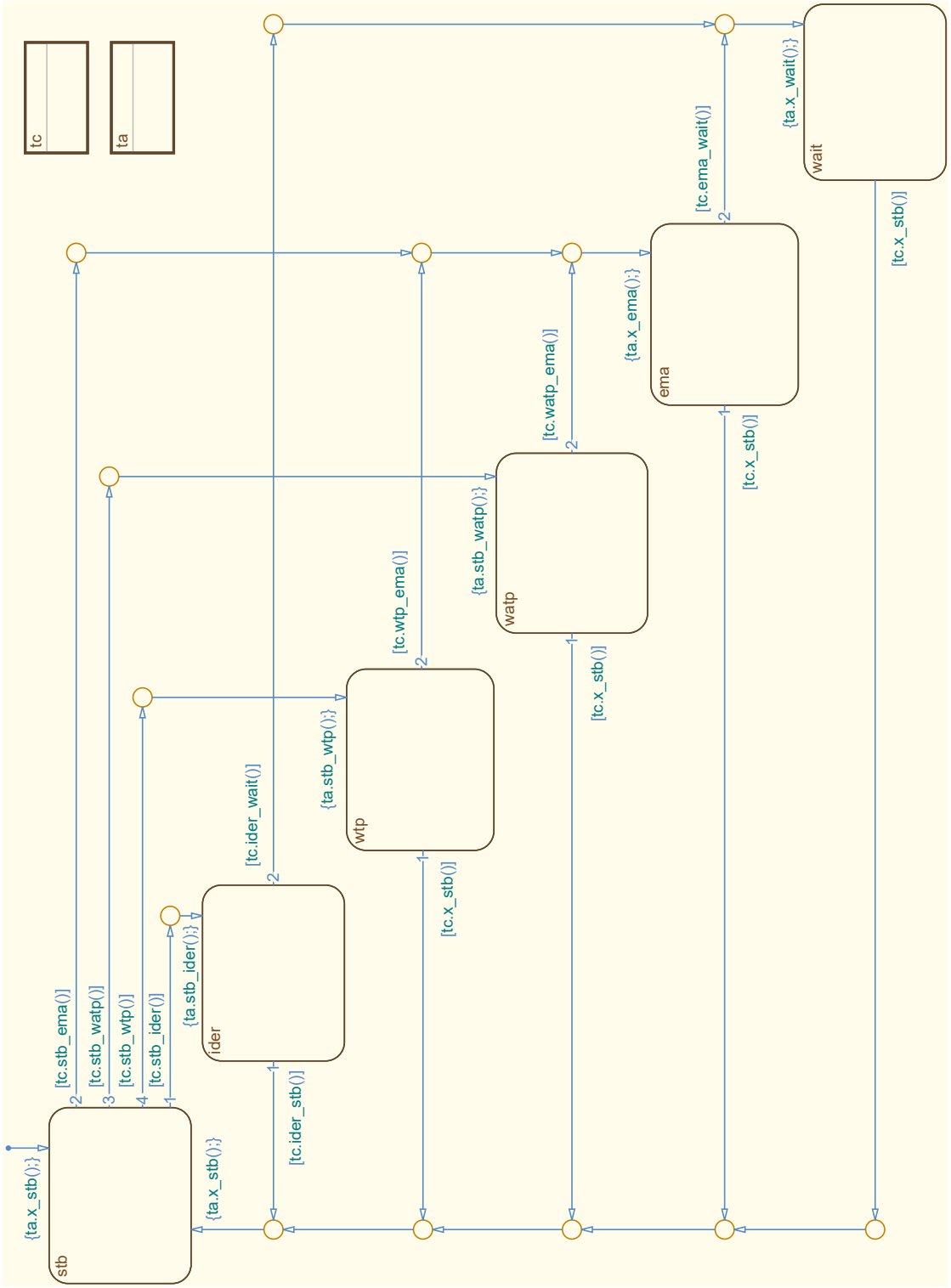


Figure 5.15: *State Machine*

Table 5.1: *Interface*

Name	Direction	Datatype	Range
`activate_flg`	input	boolean	0, 1
`useAutoTrim_flg`	input	boolean	0, 1
`waitforTrimpoint_flg`	input	boolean	0, 1
`listIdx_cfg`	input	boolean	0, 1
`listIdxValid_flg`	input	boolean	0, 1
`manGen_lgx`	input	enum	0, 1, 2, 3, 99
`trimPointOK_flg`	input	boolean	0, 1
`level1_lgx`	output	enum	0, 1, 2, 3, 4, 5, 6
`resetTP_flg`	output	boolean	0, 1

The execution of a maneuver can be aborted by setting `activate_flg` to `false`. In this case, the transition *ema→stb* is executed and *STB* is activated. This immediately stops the execution of the maneuver by setting `level1_lgx` to the respective value.

If the maneuver is completed and *WAIT* is activated there is only one exiting path. This transition, *wait→stb* to *STB*, is executed as soon as the inconsistency between the aircraft and GCS is resolved. The `activate_flg` is set by the FO via a latching switch or something similar. Therefore the `activate_flg` is still set to `true` when the maneuver is completed. This inconsistency is resolved via *WAIT*, which is active as long as `activate_flg` is set to `true`. When it is set to `false` the transition *wait→stb* is executed, the respective value for `level1_lgx` is set, and *STB* is activated again. Therefore, the mode *WAIT* is basically used here to debounce the input signal from the FO in the GCS.

After the execution of the maneuver and resolving the inconsistency, the state machine is in *STB* again. If an invalid maneuver index is selected, which is indicated by `listIdxValid_flg` being `false`, the transition *stb→ider* is executed and *IDER* is activated. This sets `level1_lgx` to the respective value and enables the FO to take appropriate action.

If this error is resolved, the transition *ider→stb* is executed. For this to take place, a valid index must be selected. Additionally, however, the `activate_flg` must also still be `false`. If both conditions are met, *STB* is activated again, which is reported to the FO by setting `level1_lgx` to the respective value.

Another exiting transition from *IDER* is possible, which is executed if the index error is resolved, but the `activate_flg` has been set to `true` beforehand. In this case, *ider→wait* is triggered and *WAIT* is activated. From there, the already known transition can be activated to reach *STB* and to resolve the inconsistency.

Table 5.2: *Transition Matrix*

		To					
		STB	IDER	WTP	WATP	EMA	WAIT
From	init	*init→stb*	n/a	n/a	n/a	n/a	n/a
	STB	-	*stb→ider*	*stb→wtp*	*stb→watp*	*stb→ema*	n/a
	IDER	*ider→stb*	-	n/a	n/a	n/a	*ider→wait*
	WTP	*wtp→stb*	n/a	-	n/a	*wtp→ema*	n/a
	WATP	*watp→stb*	n/a	n/a	-	*watp→ema*	n/a
	EMA	*ema→stb*	n/a	n/a	n/a	-	*ema→wait*
	WAIT	*wait→stb*	n/a	n/a	n/a	n/a	-

The advanced modes, which examine the individual trim point of the maneuver can also be triggered from *STB*. The transition *stb→wtp* is executed when the `activate_flg` is set to `true`, while `waitforTrimpoint_flg` is `true` as well. In this case, *WTP* is activated, which leads to the respective `level1_lgx`, but also to the deactivation of `resetTP_flg`.

Since the trim point evaluation module is no longer reset it uses the trim point information of the currently active maneuver to calculate `trimPointOK_flg`. If the altitude, heading, and speed of the aircraft match the respective trim point, the transition *wtp→ema* is executed, which activates *EMA* and starts the maneuver.

While in *WTP* the planned maneuver can be aborted. The transition *wtp→stb*, to *STB*, is executed if the `activate_flg` is set to `false`. Therefore, *STB* is activated again either directly or via *EMA* with already known transitions. When activating *STB*, `resetTP_flg` is set to `true` again, which disables the trim point calculation.

The most advanced mode of the FTMI uses not only the individual trim point data of the maneuver, but also utilizes flight control modules to automatically capture this point. If all commands from the FO in the GCS, `activate_flg`, `waitforTrimpoint_flg`, and `useAutoTrim_flg` are set to `true`, the transition *stb→watp* is executed. This sets `level1_lgx` to the respective value, enables the trim point detection by setting `resetTP_flg` to `true`, and activates *WATP*. The state execution module of the *Execution* part then enables the AFCS and IL. The trim point values of the maneuver are forwarded as commands to those modules, which lead the aircraft to the desired point.

Reaching the desired trim point is indicated by `trimPointOK_flg`, which triggers *watp→ema*. In this case, *EMA* is activated, which represents the last step in the automatic capture and evaluation of the trim point with successive maneuver execution.

The capture of the trim point can be aborted at any time by setting `activate_flg` to `false`. This triggers *watp→stb* and activates *STB*, which returns the FTMI to its default state from where the next maneuver can be selected and activated again.

5.5 Maneuvers

This section introduces the currently implemented maneuvers, that can be used for various tasks like identifying Flight Dynamic Models (FDMs), evaluate controller performance, and analyzing actuator tracking. Those maneuvers are used within the FTMI but are not designed or implemented by the author. They are shown here to create a better understanding of the overall concept. A list of the available maneuvers, which are implemented within the *Maneuver Generation* (c.f. Subsection 5.1.3), is provided in the following.

- *Blank*
- *Multi-Step*
- *Multi-Ramp*
- *Multi-Sine*
- *Sweep*
- *Spline*

The maneuver *Blank* is, as the name suggests, not a real maneuver. However, it is necessary for two reasons. On the one hand, it can be used to better organize the list of maneuvers, e.g. a new group of specific maneuvers can start with a multiple of ten or a hundred. On the other hand, it can be used to test the engagement and disengagement of the module without generating a non-zero output. While *Multi-Step* or *Multi-Ramp* maneuvers can at least theoretically be performed by test pilots, the *Multi-Sine*, *Sweep*, and *Spline* maneuvers are constructed of complex signals, that cannot be replicated by human test pilots.

The other five, non-blank, maneuvers will be described in the following in more detail, including example plots. Besides the signal shapes, the figures also show the signal traces for the `active_flg` and `status_lgx`. While the `active_flg` is the command from the state machine that corresponds to *EMA*, `status_lgx` is the internal status logic of the *Maneuver Generation* module, which is also sent to the FO in the GCS. The signals are normalized to the maximum surface deflection and therefore have no unit.

For each maneuver, up to five internal signals can be defined by the user and mapped to the respective outputs. This allows the user to use support functions with secondary signals. If one signal is used to e.g. generate a maneuver on the elevator, another signal can be used to specify the target bank-angle for this maneuver. With such a generic mapping, support functions like holding the wings level or flying with a specified load-factor can easily be activated by the user. The online scaling of amplitudes is implemented for simulation and testing but deactivated during flight tests for safety reasons. Additionally, for most maneuvers, a repetition count can be used to generate highly customizable maneuvers.

5.5.1 Multi-Step

The *Multi-Step* maneuver is a classical approach that is used to collect data for parameter estimation. It excites the system using step inputs with varying times. [KM2006, Jat2006]

Commonly used variations are `3-2-1-1`, `1-2-1`, `2-1-1`, and doublet maneuvers. The numbers in the naming schemes describe the relation of step times with respect to a base step width. A `1-2-1` maneuver together with a time-skewed doublet, both of which are executed in the lateral plane, is shown as an example in Figure 5.16. In this case, the `1-2-1` is commanded to the aileron (`xi`), while the doublet is commanded to the rudder (`zeta`). Such a maneuver can be used to excite the roll mode and dutch roll motion.

Adjustable parameters for this maneuver include the base step width, step sequence with a multiplier, and amplitude. Additionally, a repetition count can be used to automatically repeat the maneuver.

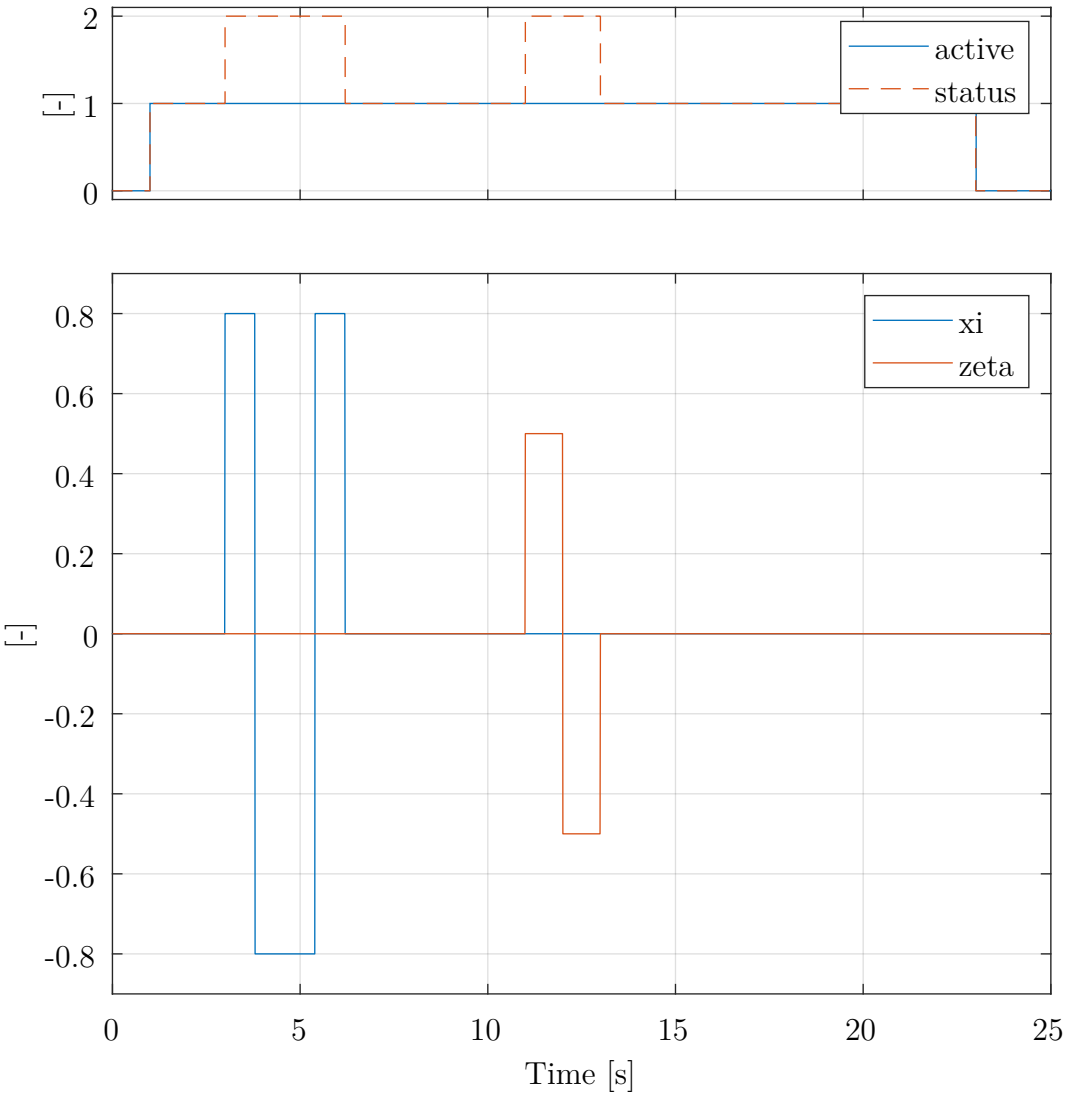


Figure 5.16: *Multi-Step Maneuver*

5.5.2 Multi-Ramp

If the slope of the rising or falling edge of the signal needs to be gradual, the *Multi-Ramp* maneuver can be utilized. Figure 5.17 shows an example where a signal is mapped to both throttle levers (`delta_t`) simultaneously. Such a maneuver can be, if assisted by other control loops that keep the altitude constant and wings level, used for a level acceleration maneuver. The generated data can then be used to evaluate engine performance or to investigate drag and lift characteristics over a range of velocities.

Adjustable parameters for this type of maneuver include the slope and time of the ramp, which can be utilized to generate various shapes. Similar to the previous maneuver a repetition count can be used to generate highly customizable saw-tooth sequences.

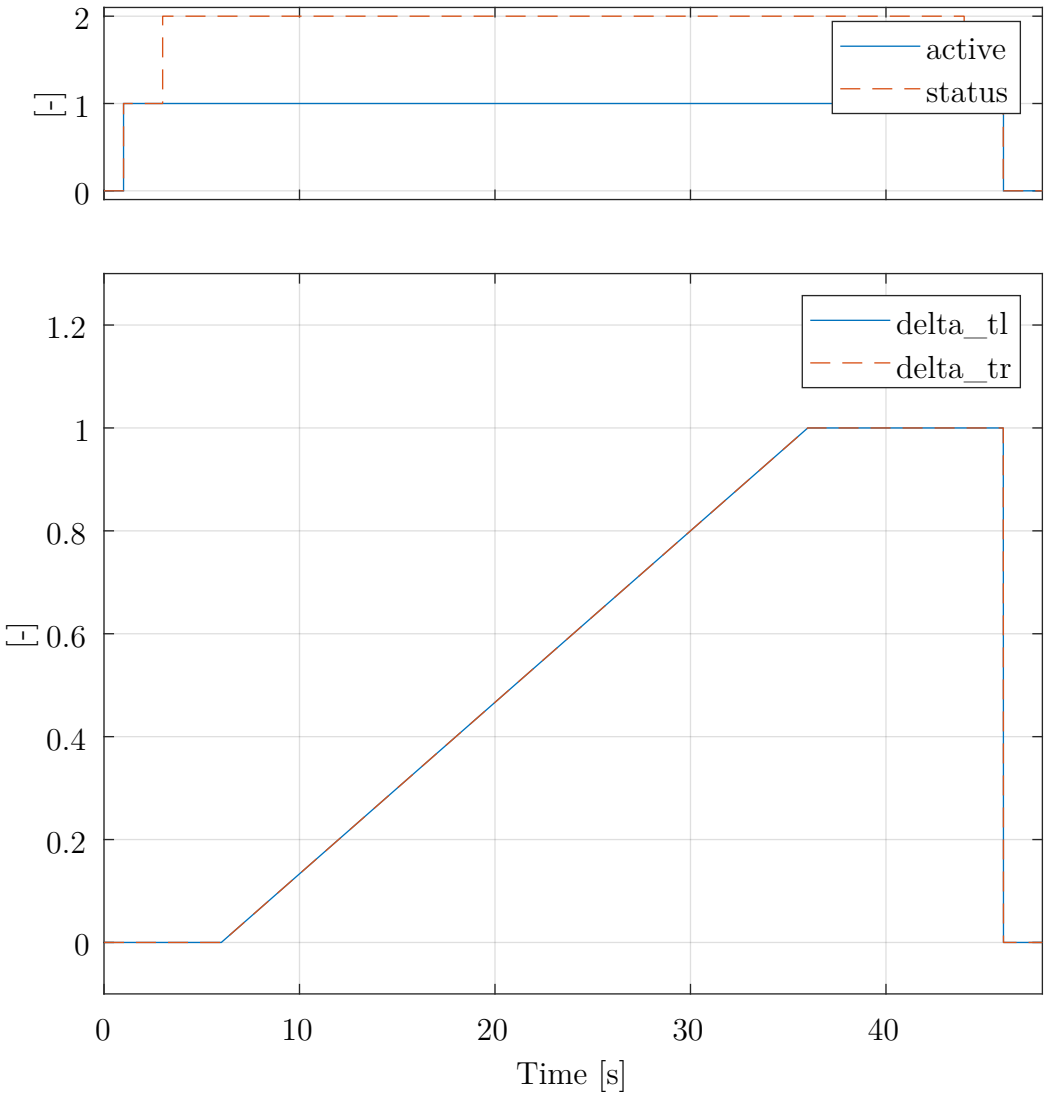


Figure 5.17: *Multi-Ramp Maneuver*

5.5.3 Multi-Sine

The *Multi-Sine* is a maneuver that is applied in the frequency domain parameter estimation. This approach is using phase-optimized multi-sine signals as illustrated by *Eugene A. Morelli*. Additionally, the optimization of phase angles in order to minimize the relative peak factor is used to generated uncorrelated signals. [Mor2012b]

Adjustable parameters include the frequency range, the number of excited harmonics, and the amplitude. Those are then optimized to avoid large peaks due to interference while keeping the desired frequency spectrum constant.

Figure 5.18 depicts an example time-series, where 16 frequencies from $0.5Hz$ to $2.0Hz$ are mapped to the aileron (`xi`) and rudder (`zeta`). It is used to excite the complete lateral dynamics of the aircraft simultaneously, but with perfectly uncorrelated signals.

Configurable parameters include the minimum and maximum frequency, as well as the number of used frequencies, duration of the maneuver, and type of optimization.

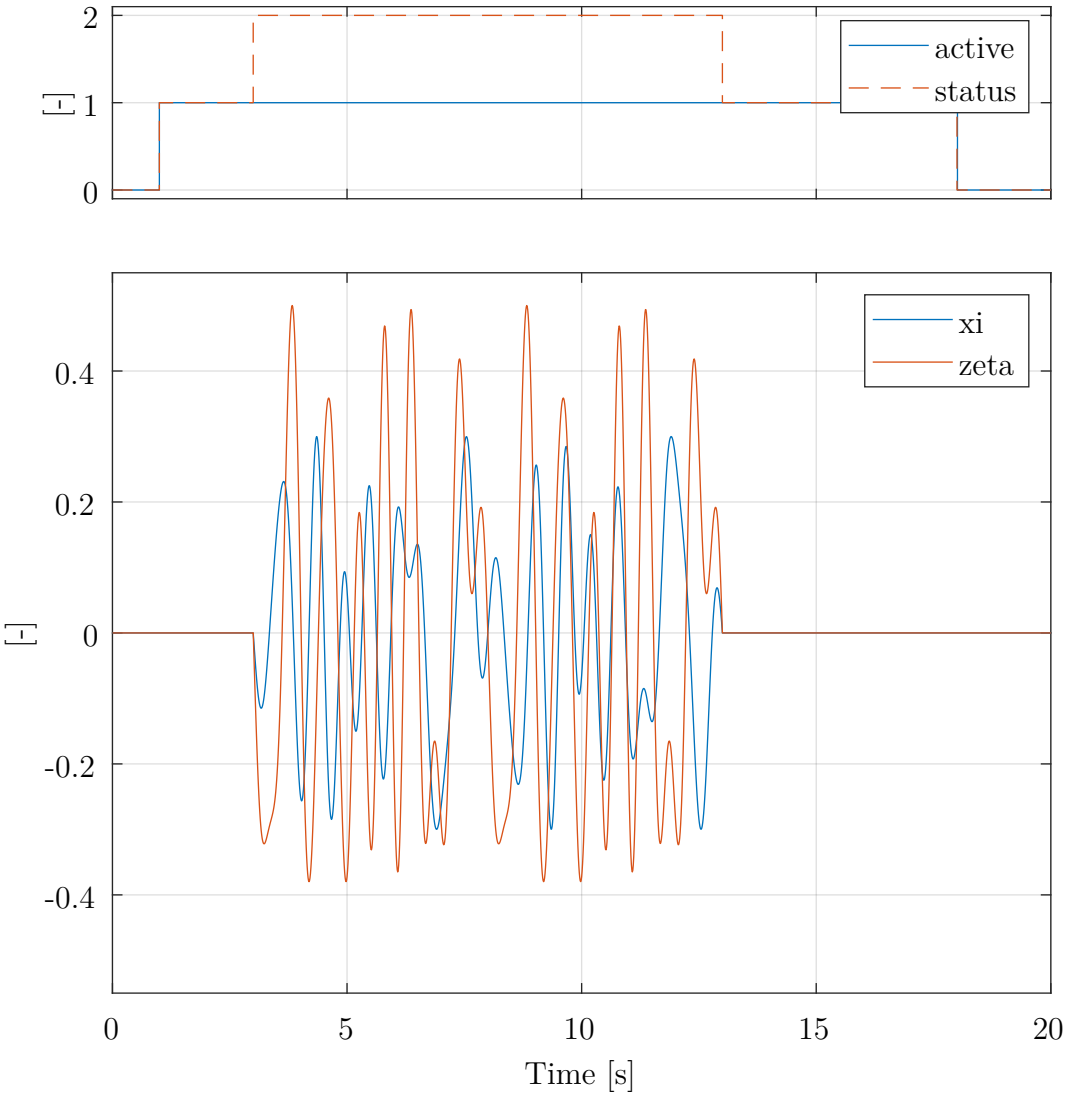


Figure 5.18: *Multi-Sine Maneuver*

5.5.4 Sweep

The *Sweep* maneuver is another approach to exited a wide range of frequencies and uses a sine wave with exponentially increasing frequency. This method, also known as logarithmic sweep, avoids the problems of linear sweeps where comparatively a large amount of time is spent at high frequencies. [TR2012]

An example of such a maneuver is depicted in Figure 5.19. In this case, only one surface, the elevator (`eta`), is used for injection. It can be seen, that the exponential increase leads to more time at longer oscillation periods. This advantage ensures that the lower frequencies are exited properly before moving to higher ones.

Configurable parameters for this maneuver include the frequency range, duration, and amplitude. Those can be used to create customized sequences for each application.

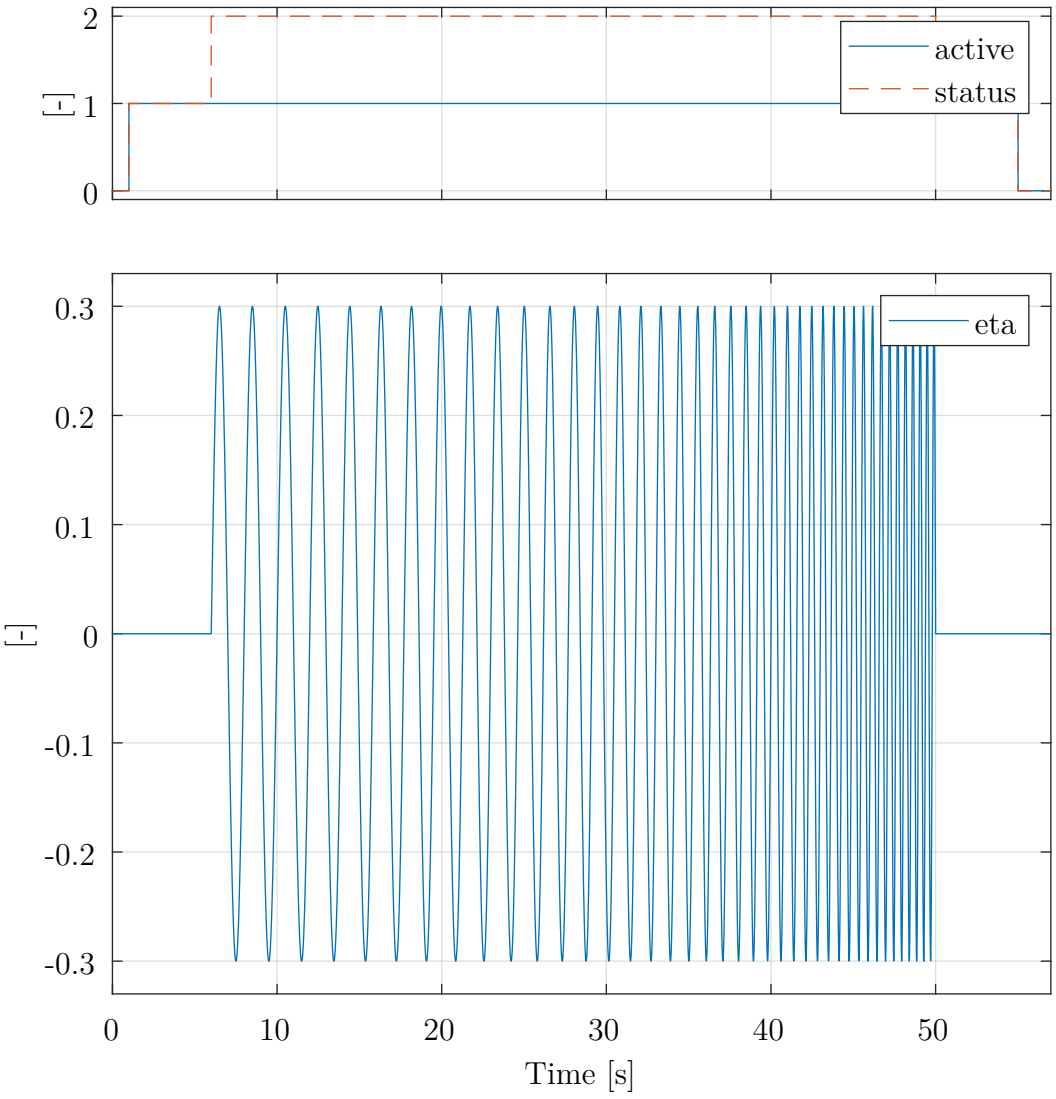


Figure 5.19: *Sweep Maneuver*

5.5.5 Spline

The *Spline* maneuver is the most customizable signal shape currently implemented. This maneuver is defined using knots, spline coefficients, and the spline order. This leads to generic and arbitrary signal shapes, that can be used if previous maneuver types are not sufficient. An example is shown in Figure 5.20.

Another use case for the *Spline* maneuver is the interface to other maneuver definitions. It can represent signals that were designed specifically to maximize the information content of the recorded data. *Eugene A. Morelli* illustrates such an approach where dynamic programming is used to maximize information [Mor1990, MK1990]. This leads to input signals that can only attain discrete values, which is not possible with the previous maneuver types. However, they can be represented by zero-order splines.

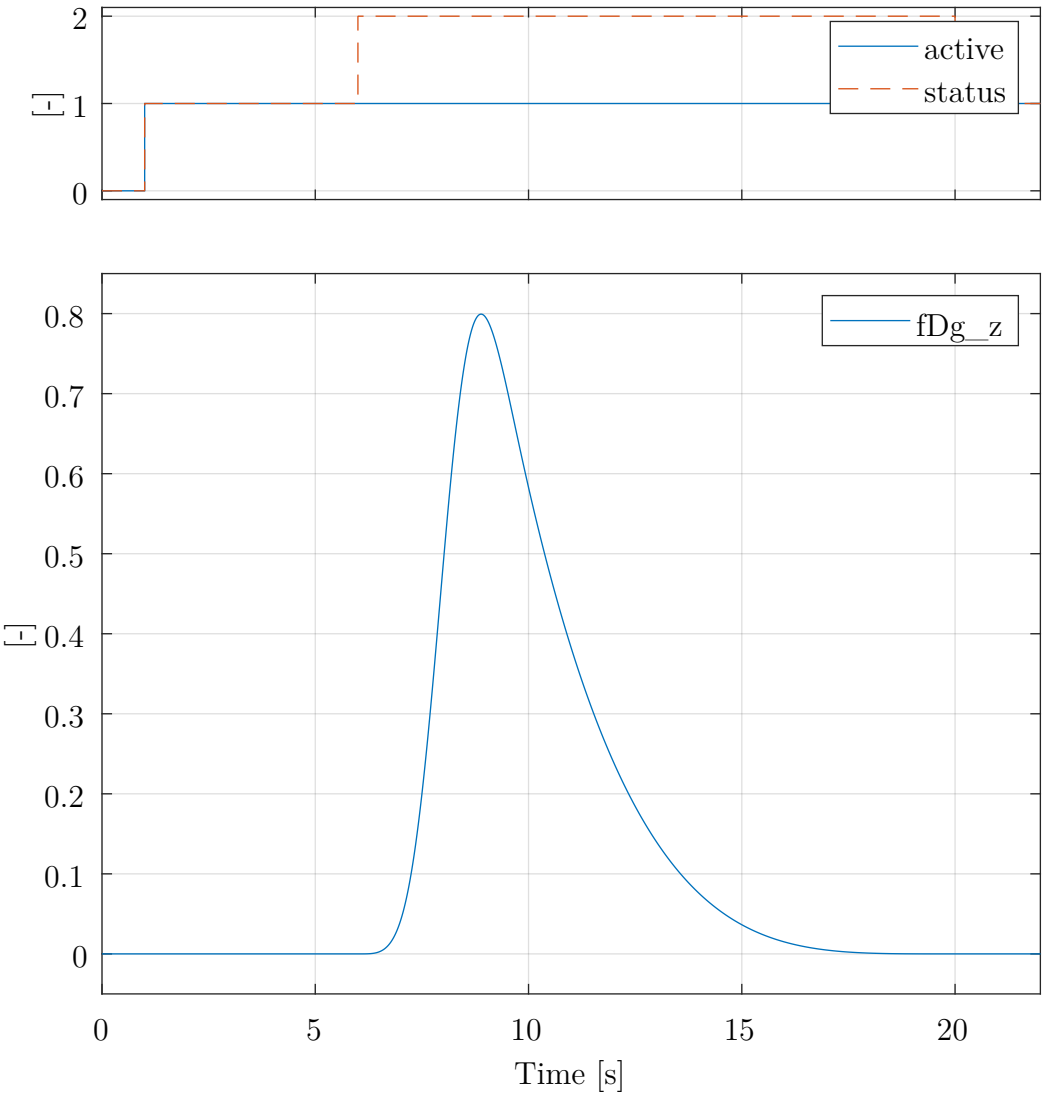


Figure 5.20: *Spline Maneuver*

5.6 Flight Tests

The FTMI has been used in various flight tests. Not so much to prove the functionality itself, but more to support the development of FDMs and flight control loops in multiple projects. This was especially true for *ELIAS* and the *Do 228*.

As introduced in Chapter 2, *ELIAS* is an ultralight OPV with a wingspan of $11m$, an MTOM of $320kg$, and a fully electric propulsion system. On the other hand, the *Do 228* is Part 23 Class IV aircraft. It is placed within the commuter category, has a wingspan of over $16m$ and an MTOM of almost $6000kg$. Examples for the real-life application of maneuver injection during flight tests are presented in the following subsections and prove the functionality of the FTMI.

The FTMI is designed with the contrary requirements of both main modules in mind. However, in the above-mentioned projects, it was used on a single computer.

However, before performing real-life flight tests, numerous testing steps have been fulfilled to gain the required confidence in the correct execution of the flight tests. As introduced in Section 3.4, those tests included Unit Tests, Model Checking, Model in the Loop (MiL), Software in the Loop (SiL), Hardware in the Loop (HiL), and Aircraft in the Loop (AiL) simulations. Additionally, various ground tests have been conducted, before moving to real-life flight tests.

Model Checking, which utilizes formal methods has been used to detect design errors, generated test case coverage reports, and prove key functionalities of the injection. This was especially useful in the state machine and logic parts of the software module. To guarantee the generation of valid maneuvers numerous SiL and HiL simulations have been executed, which especially focused on the maneuver storage within the generated code and FCC itself. In the following, maneuvers have been injected in the AiL tests before moving to Ground Tests as well as Flight Tests.

The current implementation of the FTMI supports five maneuvers, as introduced in Section 5.5. In the two above-mentioned projects, numerous *Multi-Step*, *Multi-Ramp*, *Multi-Sine*, and *Sweep* maneuvers have been used to support the development or to assist during troubleshooting.

Multi-Step maneuvers have been used with *ELIAS* to validate the FDM. Since both projects had problems with actuator performance, a *Sweep* maneuver is presented for *ELIAS* and a *Multi-Ramp* maneuver is shown for the *Do 228*. Additionally, a *Multi-Sine* maneuver with the *Do 228* is provided as an example of controller performance analysis.

The *Spline* maneuver is integrated as the most adjustable version of a signal. However, during those projects, it was not necessary since the other maneuvers included enough customization. To prove the real-life applicability of the FTMI the above-mentioned two example maneuvers of each project are presented in the following.

5.6.1 ELIAS

The ultralight aircraft *ELIAS*, introduced in Section 2.3, was the demonstration platform for the project EUROPAS and is also used in the subsequent project AURAIS. During those projects the FTMI has been extensively used for, among others, validating the FDM, identifying actuator tracking, and analyzing control loop performance. In 2019 alone, over 900 maneuvers have been executed with the FTMI, creating a total maneuver duration time of almost seven hours. In total over 300 maneuvers are defined for *ELIAS* and an assignment to the different types of maneuvers is listed in Table 5.3.

In the following, two maneuvers executed during ground trials and flight tests with *ELIAS* are shown, to prove the real-life applicability of the FTMI in an OPV. Figure 5.21 shows *ELIAS* during flight tests in May, 2017. The second maneuver is part of ground trials in 2018, in which the reconstruction of the flight mechanical system is tested and the actuator performance is validated.

Table 5.3: *ELIAS Maneuvers*

Maneuver Type	Number of defined Maneuvers
Multi-Step	185
Multi-Ramp	31
Multi-Sine	82
Sweep	7

Figure 5.21: *ELIAS in Flight at Landshut Airport (EDML) in 2017*

In Figure 5.22 a doublet, one of the *Multi-Step* maneuvers, executed by the FTMI is shown. The top part of the figure distinguishes between the command, the actuator, and the surface, which all are converted with respect to the surface. The maneuver starts shortly after the 1905s mark with five seconds in which the command reflects the trim value. In the following, the doublet with an amplitude of 0.7° is executed. This is followed by a 20s hold time of the trim value to observe the aircraft's response.

It can be seen that there is a difference between the command and the actuator, which is caused by insufficient power of the actuator. Additionally, the difference between the actuator and the surface is caused by mechanical backlash.

The maneuver is designed to stimulate the phugoid motion of the aircraft. The response of the aircraft in speed and altitude is shown in the lower part of Figure 5.22, where the expected phase-shifted response in altitude and airspeed is visible. Those are shown as the deviation (`delta`) from the trim point.

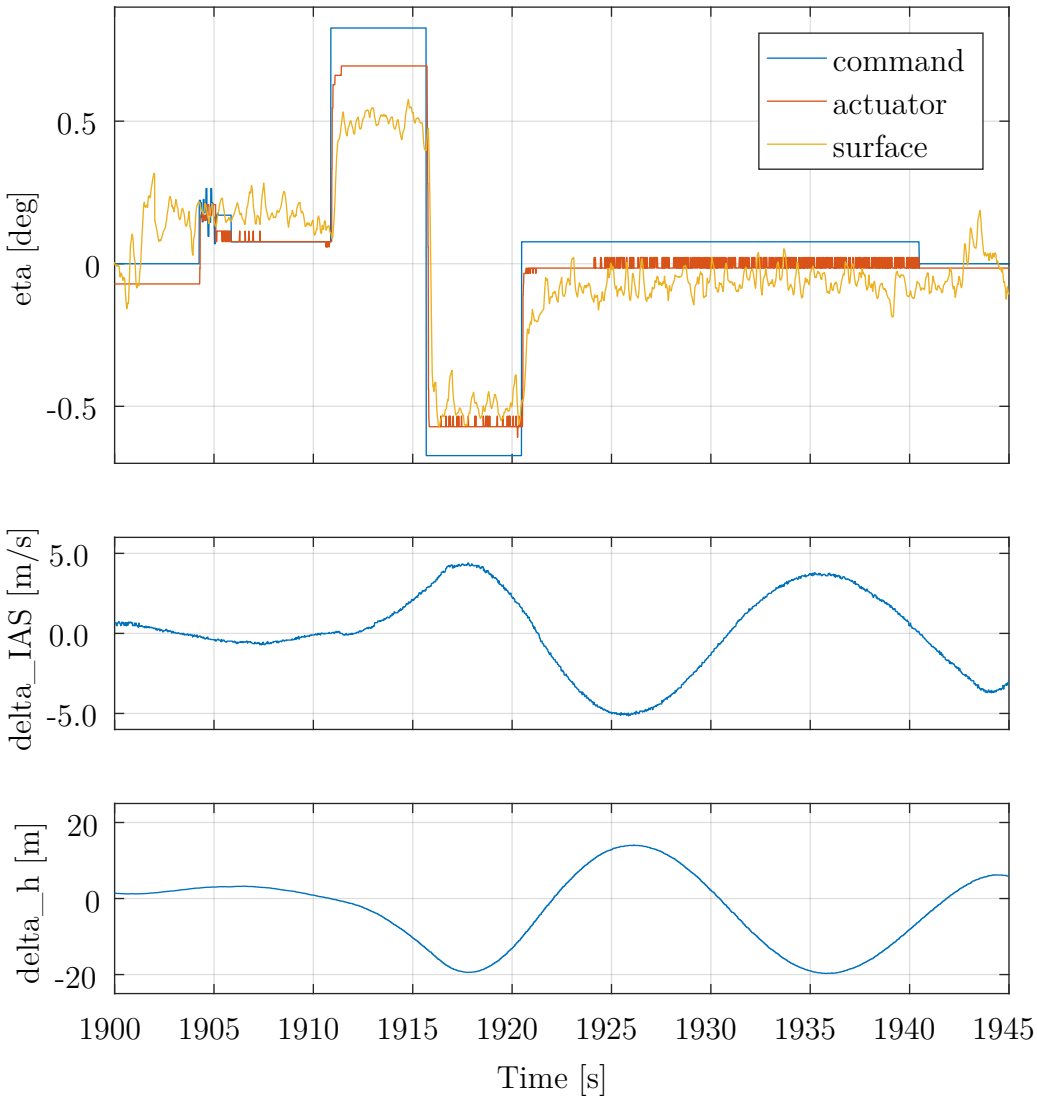


Figure 5.22: *Elevator Doublet*

A *Sweep* maneuver, used in ground trials of *ELIAS*, is shown in Figure 5.23. It shows the same maneuver being injected by the FTMI into the actuators controlling the three main control surfaces. This was part of ground trials in 2018, to measure the actuator performance in combination with the mechanical linkage to the surface.

It can be seen, that all three actuators and surfaces show similar but slightly different behavior. In the top part of Figure 5.23, the actuator of the elevator surface (`eta`) overshoots the command on positive commands but undershoots it on the opposite side. The surface is lagging behind and shows an offset as well as saturation issues. In the middle of Figure 5.23, the aileron surface (`xi`) follows its actuator more closely. However, the actuator itself creates a bigger difference to the command. In the lower part of Figure 5.23, the actuator for the rudder surface (`zeta`) lags behind the command, and the surface is following closely on negative commands but shows larger offsets during positive commands.

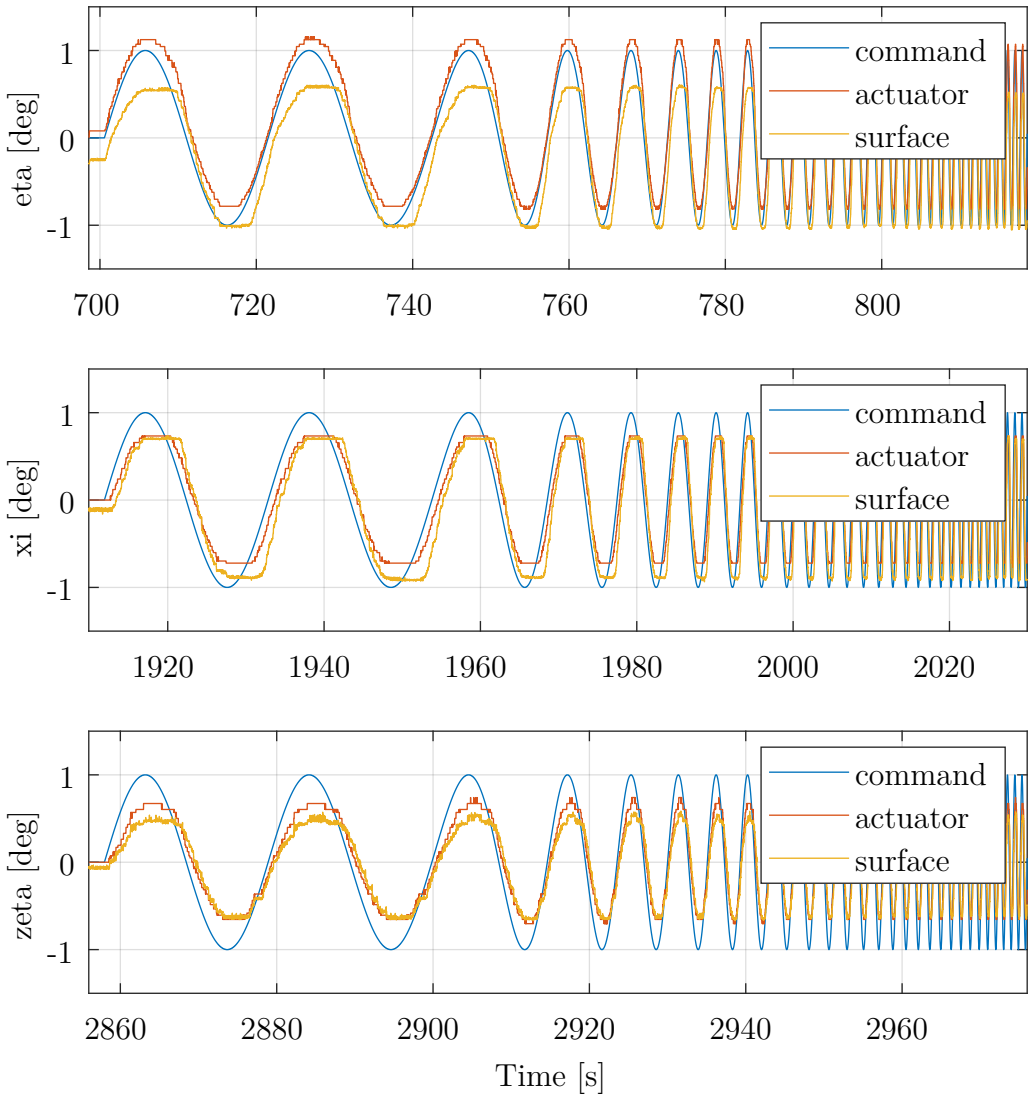


Figure 5.23: *Actuator Sweeps*

5.6.2 Do 228

The *Do 228* is the largest demonstration platform used for testing of the software modules described in this thesis. It has a wingspan of nearly $17m$, an MTOM of almost $6000kg$, and is a Part 23 Class IV, commuter category, aircraft. It is introduced and described in more detail in Section 2.4.

Over the last years, numerous ground and airborne maneuvers have been executed utilizing the FTMI. In total over 600 maneuvers have been defined for the *Do 228*. An overview of the allocation between the number of defined maneuvers and the different types is listed in Table 5.4.

Figure 5.24 shows the aircraft in flight in 2021. The following maneuvers were part of the airborne flight test campaign in 2019. They demonstrate the real-life applicability of the FTMI in a manned aircraft.

Table 5.4: *Do 228 Maneuvers*

Maneuver Type	**Number of defined Maneuvers**
Multi-Step	116
Multi-Ramp	164
Multi-Sine	293
Sweep	40

Figure 5.24: *Do 228 in Flight at Oberpfaffenhofen Airport (EDMO) in 2021*

Three *Multi-Ramp* maneuvers, generated by the FTMI are shown in Figure 5.25. Those maneuvers are injected into the aileron surface (`xi`) of the *Do 228* during one flight test, to analyze the behavior of the respective actuator and control surface linkage. All maneuvers use the current deflection of the aileron as a trim offset.

In the upper part of Figure 5.25, a ramp with an edge time of $2s$, a hold time of $1s$, and an amplitude of $0.5°$ is injected. However, basically, no reaction of the control surface can be observed.

In the next subplot, the amplitude of the ramp is doubled to $1°$. In contrast to the first subplot, it can be seen, that the surface is following the command. However, when moving from one side to the other, a lag can be identified.

The last subplot, in the bottom part of Figure 5.25, shows a ramp with a much quicker rise and fall time. It is eight times faster ($0.25s$) compared to the previous ramps. The surface is able to follow the command, at least on this accuracy and time scale.

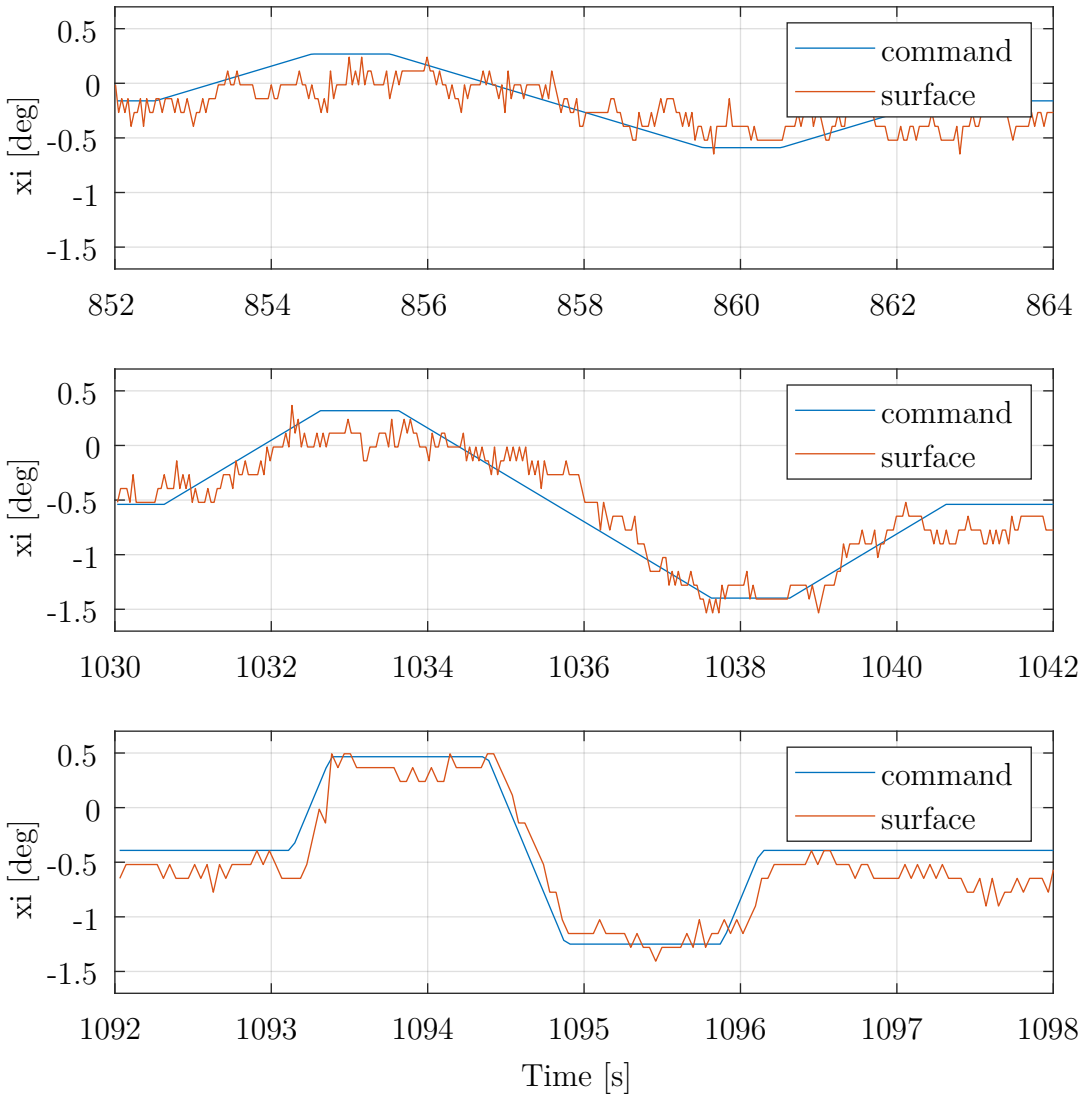


Figure 5.25: *Aileron Ramps*

A *Multi-Sine* maneuver, used with the *Do 228*, is shown in Figure 5.26. This more complex maneuver is used to investigate the performance of the IL.

The complete maneuver is actually made up of a smaller part that is repeated multiple times to increase the consistency of the information. In Figure 5.26 three of eight replications used for this maneuver are shown.

The *Multi-Sine* maneuver is injected into the lateral control part of the inner loop. The upper part, of Figure 5.26, shows status information, which can be used to locate the exact start of the maneuver. The bank-angle command (`phi`) and the response of the aircraft are shown in the bottom part. It can be seen that the maneuver starts with a zero-command time of $3s$, before actually beginning with the sine signal. Also, no trim value is used as an offset in this case. In general, the measurement is following the command with some delay and scaling issues, that were resolved afterward.

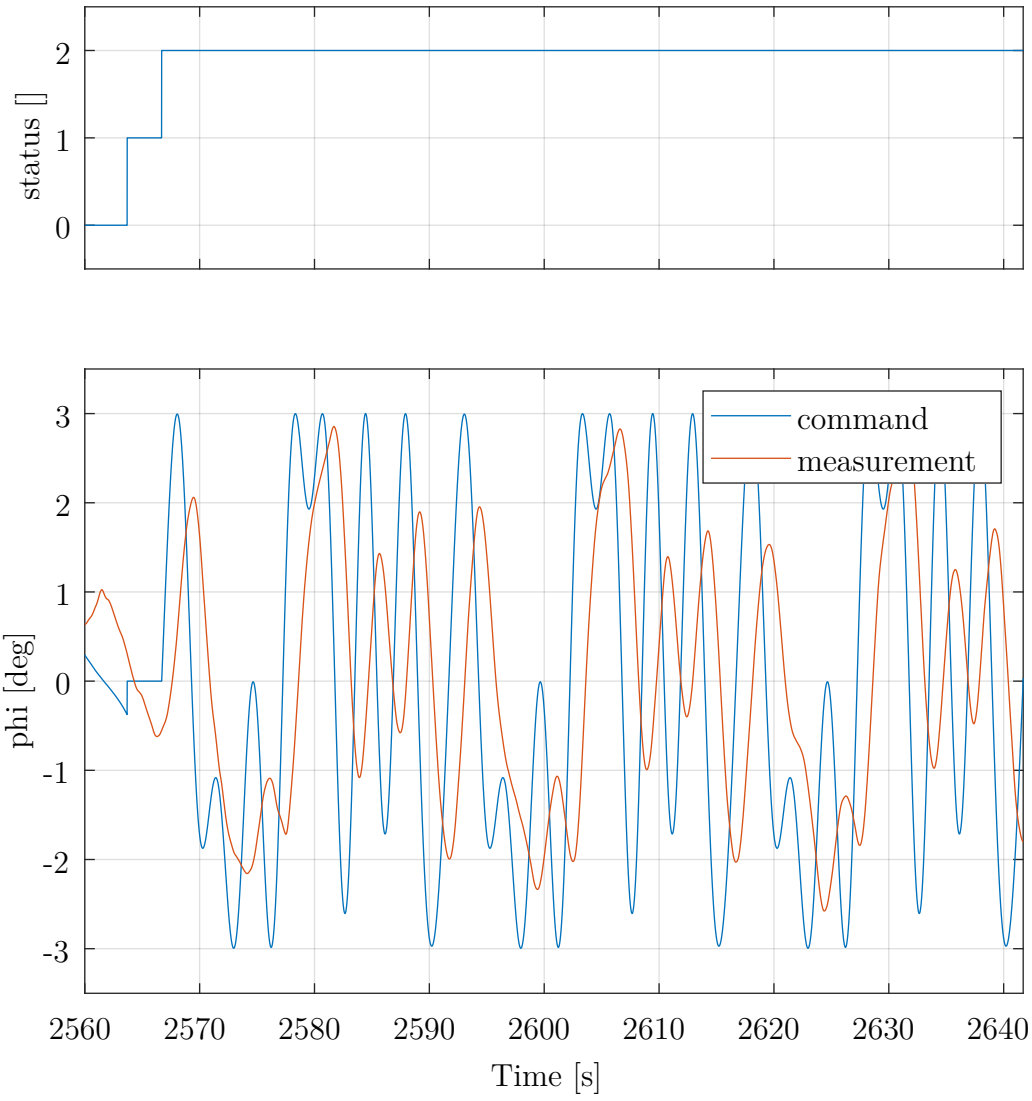


Figure 5.26: *Bank-Angle Multi-Sine*

5.7 Summary

This chapter presents the *Multi-Maneuver Multi-Control-Level Flight Test Maneuver Injection with automatic Trim Point Capture and Verification (FTMI).*

In the beginning, the hardware architecture, Flight Control Computer (FCC) system architecture, and software module architecture are presented. A differentiation is made between the two aerial platforms, *ELIAS* and the *Do 228*, used as demonstration aircraft for the FTMI. While one is an ultralight electric platform, the other is a 19-seater commuter aircraft. While the system architecture describes the integration of the FTMI with the other FCC flight control loops, the next part describes the different software modules within the FTMI. They include the *Input Processing*, *Data Processing*, *Trim Point Calculation*, *State Machine*, *State Execution*, *Maneuver Generation*, *Protections*, and the *Output Processing* module. This detailed description of the architecture is associated with contribution *C3.1 - Generic design pattern, providing a free maneuver parametrization without reimplementation.*

In the following, the three injection modules, which are used to feed maneuvers into the different flight control loops are introduced. They consist of an override and/or an injection part. This is based on the necessary functionality as they are placed prior to the *Auto Flight Control System (AFCS)*, the *Inner Loop (IL)*, and the actuators to access all necessary control levels. Their functionality and integration within the FCC represent contribution *C3.2 - Dynamic flexible choice of injection points on multiple control levels, enabling a generic implementation and safe execution.*

In the next two sections, the operation modes, as well as the transition conditions and transition actions of the main state machine, are presented. The available operation modes are *Standby* (*STB*), *Index-Error* (*IDER*), *Wait-for-Trim-Point* (*WTP*), *Wait-for-Auto-Trim-Point* (*WATP*), *Execute-Maneuver* (*EMA*), and *Wait* (*WAIT*). This state machine and its connections to the other flight control modules enable the advanced features of the FTMI and constitutes contribution *C3.3 - Individual trim point verification and automatic trim point capture for safe and effective flight testing.*

Additionally, the available maneuvers are presented. While those are not developed by the author, they are included in this thesis to provide a better understanding of the overall functionality of the FTMI. Those five maneuvers are *Multi-Step*, *Multi-Ramp*, *Multi-Sine*, *Sweep*, and *Spline*.

This is followed by a section with real-life flight test data. Both aerial test platforms, the Optionally-Piloted Vehicle (OPV) *ELIAS* and the manned *Do 228* have performed numerous flight tests. During those flights, from which a few are presented, the functions of the FTMI have been used to validate Flight Dynamic Model (FDM), investigate IL performance, or analyze actuator tracking. This proofs the real-life applicability of the contributions and the FTMI in general. Additionally, it proves the applicability of the methodology, presented in Chapter 3, since it was used to design, implement, and test this software module.

Chapter 6

Conclusion

In this thesis, a methodology to design, implement, and test higher-level system automation functions is proposed. Additionally, two applications based on this methodology are introduced. Those are a flight control system automation for experimental aircraft and a flight test maneuver injection. The conclusion for all three is presented in the following sections: Section 6.1, Section 6.2, and Section 6.3.

A key requirement for the developed algorithms is the real-life applicability to various types of aircraft. These include Unmanned Aerial Vehicles (UAVs), Optionally-Piloted Vehicles (OPVs), and manned aircraft. To verify this requirement, the applications presented in this thesis are tested on multiple demonstration platforms. An overview of all aircraft is given in Figure 6.1.

(a) SAGITTA - UAV [Air2017]

(b) DA 42 - OPV

(c) ELIAS - OPV

(d) Do 228 - manned

Figure 6.1: *Aerial Demonstration Platforms, referenced in this thesis*

6.1 Methodology for System Automation

A generic methodology for higher-level system automation and its formal verification, called *Design, Implementation, and Testing Methodology for System Automation - using State Machines in Stateflow (MTSA)*, is developed to create the basis for designing and implementing higher-level automation functions. It is based on dividing functions into different modes of operation and assigning those to their respective level of automation. The connection to other systems is purely handled in external interaction points and switches. To prove the correct functionality of the design and the implemented automation, the methodology also includes formal testing and verification.

The function to be automated is divided into different modes with respect to commonly used modules or parts of the existing flight control loops. Depending on the number of modes and different levels of control involvement they can be grouped into common levels of automation. Changes from one mode to another are restricted by transition conditions. Those can be inputs from the user, conditions for an automatic sequence of modes, or guard conditions to restrict usage of specific modes in certain aircraft or environments. Afterward, if the transition condition is met, the respective transition action is executed and the new mode is activated.

The connection to other modules is not designed to pass through the main automation module, but instead, the commands are induced via external interaction points and switches. This modular approach allows for a generic implementation that is flexible with respect to future extension or modification. The higher-level automation proposed as targets for this methodology are advanced finite state machines.

Testing the design and implementation of the function is a fundamental development step for safety-critical software. Therefore, this methodology includes a process for formal testing and verification. This covers not only closed-loop tests but also independent tests using formal methods. Consequently, this methodology leads to deterministic and predictable robust automation with formally guaranteed properties.

The contributions related to the Methodology for System Automation, which are presented in Chapter 3, are summarized in the following.

Contributions - Methodology for System Automation

- C1.1 - Hierarchical decomposition design strategy, minimizing complexity and optimizing testability
- C1.2 - Modeling guidelines for implementation, minimizing opacity and maximizing software maintainability
- C1.3 - Incremental bottom-up application of formal methods, ensuring effective testing and guaranteed system characteristics

6.2 Flight Control System Automation

The *Operator-Centric Multi-User Flight Control System Automation for experimental UAVs and OPVs with Contingency Procedures (FCSA)*, developed in this thesis, is a software module that is part of the Flight Control Computer (FCC). It provides an interface for multiple users, like a Flight Operator (FO) in a Ground Control Station (GCS) and an External Pilot (EP). Additionally, the FCSA administers a cascaded control loop consisting of various flight control modules to create numerous dedicated flight modes.

Overall, 15 control modes and four superposition options are spread over five levels. For the FO those include, among others, fully automatic takeoff and landing, trajectory modes like GPS-based waypoint flight, common autopilot-based modes, like altitude, heading, and speed command, and very basic modes like parking and standby. Example control modes for the EP are dedicated ground and airborne modes, like direct-law or rate-command-attitude-hold, which are switched automatically based on the state of the aircraft. The design makes it possible to use them on a UAV as well as on an OPV, without changing the internal structure. Furthermore, the FO and EP, can both be either onboard the aircraft or in a remote location, and superposition options can be used.

Besides the nominal functions, it features contingency procedures, both for operational as well as malfunction scenarios. If the user wants to "cancel" or "pause" the mission, modes guiding the aircraft back to the home base, or performing automatic holding patterns can be activated. If the system detects a malfunction it will automatically activate corresponding contingency modes, to mitigate the effects. Possible faults include a disconnected control link or a loss of the Global Positioning System (GPS) signal.

The FCSA was developed for and tested on a UAV, the *SAGITTA* Research Demonstrator, and on an OPV, the *DA 42*. Platform type-specific modes, e.g. modes only for a UAV or OPV, are guarded so they cannot be activated on the wrong platform. Numerous flight tests on both platforms have proven the real-life applicability of the FCSA on both platforms in various conditions.

The contributions related to the Flight Control System Automation, which are presented in Chapter 4, are summarized in the following.

Contributions - Flight Control System Automation

- C2.1 - Strategy for switchability between various modes on different authority levels, enabling experimental automation
- C2.2 - Operational management concept for multi-user experimental OPVs and UAVs, increasing mode awareness
- C2.3 - Automatic operational and malfunction contingency procedures for continuous operation in non-nominal circumstances

6.3 Flight Test Maneuver Injection

The *Multi-Maneuver Multi-Control-Level Flight Test Maneuver Injection with automatic Trim Point Capture and Verification (FTMI)*, developed in this thesis, is a software module that is capable of injecting various maneuvers into different points of the cascaded control loop for fully automatic flight test execution.

This automation is designed in a modular fashion, separating parts based on their individual requirements. Therefore, the FTMI is split into two main parts. While one of them relies on storage capacity, the other requires real-time capabilities. Both are connected over an asynchronous interface to support execution on different systems.

Besides the main data that defines the maneuver itself, each of them also includes a trim point, which in turn consists of up to six parameters. Depending on the configuration of each maneuver the FTMI can either ignore this trim point, which results in the immediate execution of the maneuver after activation, or it analyses the current state of the aircraft and holds the execution until the trim point is reached. Furthermore, it can even use existing parts of the control loop to automatically capture the trim point, without any intervention by the pilot, and start the maneuver accordingly.

The FTMI can be used to generate five different types of maneuvers. These include a *Multi-Step*, *Multi-Ramp*, *Multi-Sine*, *Sweep*, and *Spline* maneuver that are highly customizable to support various demands.

Each maneuver can be mapped to one or more of eight injection points. Those include direct actuator commands for aileron, elevator, rudder, thrust commands for up to two engines, as well as low-level inner loop commands like bank-angle, vertical load-factor, and lateral load-factor. In combination with the five maneuvers, this allows for a very large variety of different, fully automatic, flight tests.

The FTMI has been used extensively on two of the aerial demonstration platforms. With those precise and automated tests, the FTMI has proven its real-life applicability and has contributed to the success of the respective project.

The contributions related to the Flight Test Maneuver Injection, which are presented in Chapter 5, are summarized in the following.

Contributions - Flight Test Maneuver Injection

- C3.1 - Generic design pattern, providing a free maneuver parametrization without reimplementation
- C3.2 - Dynamic flexible choice of injection points on multiple control levels, enabling a generic implementation and safe execution
- C3.3 - Individual trim point verification and automatic trim point capture for safe and effective flight testing

6.4 Outlook

In this thesis, the following methodology and two applications were presented, which are verified by real-life flight tests on multiple aerial demonstration platforms.

- *Design, Implementation, and Testing Methodology for System Automation - using State Machines in Stateflow (MTSA)*
- *Operator-Centric Multi-User Flight Control System Automation for experimental UAVs and OPVs with Contingency Procedures (FCSA)*
- *Multi-Maneuver Multi-Control-Level Flight Test Maneuver Injection with automatic Trim Point Capture and Verification (FTMI)*

Future updates of the MTSA might include a change in the way values are assigned to the output bus. The current implementation focuses on maintainability and expandability. However, with the currently used method, values can generally be overwritten in multiple places, which can lead to problems. Future releases of *MATLAB* and *Simulink* might include other ways of assigning values to data collecting structures, or even entirely different structures, that can then be utilized to avoid this problem.

The FCSA can easily be extended due to its modular structure. Future projects might include other operations modes that can be added to the current implementation. Additionally, the FCSA could be extended to be used in different types of aircraft, like transition or tiltrotor aircraft, which also depend on various mode changes during the different phases of flight.

For the FTMI, a few aspects are currently under discussion and could be integrated into future releases. For aircraft with additional control sticks, that are not mechanically connected to the surfaces, an additional input to the currently generated maneuver output could be implemented to allow for low bandwidth correction by the pilot during a maneuver. In the current version of the FTMI, there is no possibility to inject maneuvers on the autopilot level because such a function was not required in any project. However, if necessary, such an injection could be added relatively easy, due to the modular implementation of the FTMI. Concerning the implementation of the injection points, a transformation of project-specific buses to generic FTMI ones, and vice versa, might be necessary, to support completely different bus structures in other projects.

Due to the rise of UAVs and OPVs in numerous applications, the developed methodology and applications can be used in various forms. This might be by enabling the implementation of complex state machines, development of multi-user automation for experimental aircraft, or supporting efficient verification and validation with automatically executed flight tests.

Appendix A

Automation Levels

Table A.1, Table A.2, and Table A.3 depict the original version of different levels of automation for manned flight, as defined by *Charles Billings* [Bil1991, p. 27]. Due to size and format issues, the original table is split up into three separate ones.

Table A.1: *Automation Levels - Management Modes*

Very Low ← Level of Automation → Very High	**Management Mode**	Very High ← Level of Involvement → Very Low
	Autonomous Operation	
	Management by Exception	
	Management by Consent	
	Management by Delegation	
	Shared Control	
	Assisted Manual Control	
	Direct Manual Control	

Table A.2: *Automation Levels - Automation Functions*

Management Mode	**Automation Functions**
Autonomous Operation	Fully autonomous operation Pilot not usually informed System may or may not be capable of being disabled
Management by Exception	Essentially autonomous operation Automatic reconfiguration System informs pilot and monitors responses
Management by Consent	Full automatic control of aircraft and flight Intent, diagnostic and prompting functions provided
Management by Delegation	Autopilot and Autothrottle control of flight path Automatic communications and nav following
Shared Control	Enhanced control and guidance Smart advisory systems Potential flight path and other predictor displays
Assisted Manual Control	Flight director, FMS, Nav modules Data link with manual messages Monitoring of flight path control and aircraft systems
Direct Manual Control	Normal warnings and alerts Voice communication with ATC Routine ACARS communications performed automatically

Table A.3: *Automation Levels - Human Functions*

Human Functions	**Management Mode**
Pilot generally has no role in operation Monitoring is limited to fault detection Goals are self-defined, pilot normally has no reason to intervene	Autonomous Operation
Pilot informed of system intent Must consent to critical decisions May intervene by reverting to lower level of management	Management by Exception
Pilot must consent to state changes, checklist execution, anomaly resolution Manual execution of critical actions	Management by Consent
Pilot commands altitude, heading, speed Manual or coupled navigation Commands system operations, checklists, communications	Management by Delegation
Pilot in control through CWS or envelope-protected system May utilize advisory systems System management is manual	Shared Control
Direct authority over all systems Manual control, aided by F/D and enhanced navigation display FMS is available, trend info on request	Assisted Manual Control
Direct authority over all systems Manual control utilizing raw data Unaided decision-making; Manual communications	Direct Manual Control

Appendix B

Edge Detector Code Generation

In Table B.1 all automatically generated files (except "*.txt" and "*.bat") are listed. The most important ones are emphasized in italics and are shown in the following.

Table B.1: *Code Generation - Files*

Group	**Name**	**Size**
C Source	*edgeDetector.c*	9 KB
	rt_nonfinite.c	2 KB
	rtGetInf.h	4 KB
	rtGetNaN.c	3 KB
DMR-File	codedescriptor.dmr	179 KB
C/C++ Header	builtin_typeid_types.h	2 KB
	edgeDetector.h	5 KB
	edgeDetector_private.h	2 KB
	edgeDetector_types.h	1 KB
	multiword_types.h	18 KB
	rt_nonfinite.h	2 KB
	rtGetInf.c	2 KB
	rtGetNaN.h	1 KB
	rtmodel.h	1 KB
	rtwtypes.h	2 KB
MAT-file	buildInfo.mat	13 KB
	codeInfo.mat	4 KB
	rtwtypeschksum.mat	2 KB
Makefile	edgeDetector.mk	14 KB
RSP-File	edgeDetector_ref.rsp	0 KB
TMW-File	rtw_proj.tmw	1 KB

```
/*
 * edgeDetector.h
 *
 * Academic License - for use in teaching, academic research, and
   meeting
 * course requirements at degree granting institutions only.  Not for
 * government, commercial, or other organizational use.
 *
 * Code generation for model "edgeDetector".
 *
 * Model version              : 1.26
 * Simulink Coder version : 8.11 (R2016b) 25-Aug-2016
 * C source code generated on : Sun Jun 07 16:27:25 2020
 *
 * Target selection: grt.tlc
 * Note: GRT includes extra infrastructure and instrumentation for
   prototyping
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objective: Debugging
 * Validation result: Not run
 */

#ifndef RTW_HEADER_edgeDetector_h_
#define RTW_HEADER_edgeDetector_h_
#include <float.h>
#include <string.h>
#include <stddef.h>
#ifndef edgeDetector_COMMON_INCLUDES_
# define edgeDetector_COMMON_INCLUDES_
#include "rtwtypes.h"
#include "rtw_continuous.h"
#include "rtw_solver.h"
#include "rt_logging.h"
#endif                                 /* edgeDetector_COMMON_INCLUDES_
    */

#include "edgeDetector_types.h"

/* Shared type includes */
#include "multiword_types.h"
#include "rt_nonfinite.h"

/* Macros for accessing real-time model data structure */
#ifndef rtmGetFinalTime
# define rtmGetFinalTime(rtm)          ((rtm)->Timing.tFinal)
#endif

#ifndef rtmGetRTWLogInfo
```

```
46 # define rtmGetRTWLogInfo(rtm)          ((rtm)->rtwLogInfo)
47 #endif
48
49 #ifndef rtmGetErrorStatus
50 # define rtmGetErrorStatus(rtm)         ((rtm)->errorStatus)
51 #endif
52
53 #ifndef rtmSetErrorStatus
54 # define rtmSetErrorStatus(rtm, val)    ((rtm)->errorStatus = (val))
55 #endif
56
57 #ifndef rtmGetStopRequested
58 # define rtmGetStopRequested(rtm)       ((rtm)->Timing.stopRequestedFlag)
59 #endif
60
61 #ifndef rtmSetStopRequested
62 # define rtmSetStopRequested(rtm, val) ((rtm)->Timing.stopRequestedFlag
      = (val))
63 #endif
64
65 #ifndef rtmGetStopRequestedPtr
66 # define rtmGetStopRequestedPtr(rtm)    (&((rtm)->Timing.
      stopRequestedFlag))
67 #endif
68
69 #ifndef rtmGetT
70 # define rtmGetT(rtm)                   ((rtm)->Timing.taskTime0)
71 #endif
72
73 #ifndef rtmGetTFinal
74 # define rtmGetTFinal(rtm)              ((rtm)->Timing.tFinal)
75 #endif
76
77 /* Block states (auto storage) for system '<Root>' */
78 typedef struct {
79   uint8_T is_active_c3_edgeDetector;    /* '<Root>/edgeDetector' */
80   uint8_T is_c3_edgeDetector;           /* '<Root>/edgeDetector' */
81 } DW_edgeDetector_T;
82
83 /* External inputs (root inport signals with auto storage) */
84 typedef struct {
85   int8_T edgeDetector_in;               /* '<Root>/edgeDetector_in' */
86 } ExtU_edgeDetector_T;
87
88 /* External outputs (root outports fed by signals with auto storage) */
89 typedef struct {
90   int8_T edgeDetector_out;              /* '<Root>/edgeDetector_out' */
91 } ExtY_edgeDetector_T;
```

```
92
93 /* Real-time Model Data Structure */
94 struct tag_RTM_edgeDetector_T {
95   const char_T *errorStatus;
96   RTWLogInfo *rtwLogInfo;
97
98   /*
99    * Timing:
100   * The following substructure contains information regarding
101   * the timing information for the model.
102   */
103  struct {
104    time_T taskTime0;
105    uint32_T clockTick0;
106    uint32_T clockTickH0;
107    time_T stepSize0;
108    time_T tFinal;
109    boolean_T stopRequestedFlag;
110  } Timing;
111 };
112
113 /* Block states (auto storage) */
114 extern DW_edgeDetector_T edgeDetector_DW;
115
116 /* External inputs (root inport signals with auto storage) */
117 extern ExtU_edgeDetector_T edgeDetector_U;
118
119 /* External outputs (root outports fed by signals with auto storage) */
120 extern ExtY_edgeDetector_T edgeDetector_Y;
121
122 /* Model entry point functions */
123 extern void edgeDetector_initialize(void);
124 extern void edgeDetector_step(void);
125 extern void edgeDetector_terminate(void);
126
127 /* Real-time Model object */
128 extern RT_MODEL_edgeDetector_T *const edgeDetector_M;
129
130 /*-
131  * The generated code includes comments that allow you to trace directly
132  * back to the appropriate location in the model.  The basic format
133  * is <system>/block_name, where system is the system number (uniquely
134  * assigned by Simulink) and block_name is the name of the block.
135  *
136  * Use the MATLAB hilite_system command to trace the generated code back
137  * to the model.  For example,
138  *
139  * hilite_system('<S3>')    - opens system 3
```

```
140  * hilite_system('<S3>/Kp') - opens and selects block Kp which resides
       in S3
141  *
142  * Here is the system hierarchy for this model
143  *
144  * '<Root>' : 'edgeDetector'
145  * '<S1>'   : 'edgeDetector/edgeDetector'
146  */
147 #endif                                 /* RTW_HEADER_edgeDetector_h_ */
```

Listing B.1: *Edge Detector - EdgeDetector.h*

```
/*
 * edgeDetector_private.h
 *
 * Academic License - for use in teaching, academic research, and
   meeting
 * course requirements at degree granting institutions only.  Not for
 * government, commercial, or other organizational use.
 *
 * Code generation for model "edgeDetector".
 *
 * Model version              : 1.26
 * Simulink Coder version : 8.11 (R2016b) 25-Aug-2016
 * C source code generated on : Sun Jun 07 16:27:25 2020
 *
 * Target selection: grt.tlc
 * Note: GRT includes extra infrastructure and instrumentation for
   prototyping
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objective: Debugging
 * Validation result: Not run
 */

#ifndef RTW_HEADER_edgeDetector_private_h_
#define RTW_HEADER_edgeDetector_private_h_
#include "rtwtypes.h"
#include "builtin_typeid_types.h"
#include "multiword_types.h"

/* Private macros used by the generated code to access rtModel */
#ifndef rtmSetTFinal
# define rtmSetTFinal(rtm, val)        ((rtm)->Timing.tFinal = (val))
#endif

#ifndef rtmGetTPtr
# define rtmGetTPtr(rtm)               (&(rtm)->Timing.taskTime0)
#endif
#endif                                 /*
   RTW_HEADER_edgeDetector_private_h_ */
```

Listing B.2: *Edge Detector - EdgeDetector_private.h*

```
/*
 * edgeDetector_types.h
 *
 * Academic License - for use in teaching, academic research, and
   meeting
 * course requirements at degree granting institutions only.  Not for
 * government, commercial, or other organizational use.
 *
 * Code generation for model "edgeDetector".
 *
 * Model version              : 1.26
 * Simulink Coder version : 8.11 (R2016b) 25-Aug-2016
 * C source code generated on : Sun Jun 07 16:27:25 2020
 *
 * Target selection: grt.tlc
 * Note: GRT includes extra infrastructure and instrumentation for
   prototyping
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objective: Debugging
 * Validation result: Not run
 */

#ifndef RTW_HEADER_edgeDetector_types_h_
#define RTW_HEADER_edgeDetector_types_h_

/* Forward declaration for rtModel */
typedef struct tag_RTM_edgeDetector_T RT_MODEL_edgeDetector_T;

#endif                                 /*
   RTW_HEADER_edgeDetector_types_h_ */
```

Listing B.3: *Edge Detector - EdgeDetector_types.h*

```c
/*
 * edgeDetector.c
 *
 * Academic License - for use in teaching, academic research, and
   meeting
 * course requirements at degree granting institutions only.  Not for
 * government, commercial, or other organizational use.
 *
 * Code generation for model "edgeDetector".
 *
 * Model version              : 1.26
 * Simulink Coder version : 8.11 (R2016b) 25-Aug-2016
 * C source code generated on : Sun Jun 07 16:27:25 2020
 *
 * Target selection: grt.tlc
 * Note: GRT includes extra infrastructure and instrumentation for
   prototyping
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objective: Debugging
 * Validation result: Not run
 */

#include "edgeDetector.h"
#include "edgeDetector_private.h"

/* Named constants for Chart: '<Root>/edgeDetector' */
#define edgeDetector_IN_NO_ACTIVE_CHILD ((uint8_T)0U)
#define edgeDetector_IN_S0              ((uint8_T)1U)
#define edgeDetector_IN_S0Edge          ((uint8_T)2U)
#define edgeDetector_IN_S1              ((uint8_T)3U)
#define edgeDetector_IN_S1Edge          ((uint8_T)4U)

/* Block states (auto storage) */
DW_edgeDetector_T edgeDetector_DW;

/* External inputs (root inport signals with auto storage) */
ExtU_edgeDetector_T edgeDetector_U;

/* External outputs (root outports fed by signals with auto storage) */
ExtY_edgeDetector_T edgeDetector_Y;

/* Real-time model */
RT_MODEL_edgeDetector_T edgeDetector_M_;
RT_MODEL_edgeDetector_T *const edgeDetector_M = &edgeDetector_M_;

/* Model step function */
void edgeDetector_step(void)
{
```

```
47  /* Chart: '<Root>/edgeDetector' incorporates:
48   *  Inport: '<Root>/edgeDetector_in'
49   */
50  /* Gateway: edgeDetector */
51  /* During: edgeDetector */
52  if (edgeDetector_DW.is_active_c3_edgeDetector == 0U) {
53    /* Entry: edgeDetector */
54    edgeDetector_DW.is_active_c3_edgeDetector = 1U;
55
56    /* Entry Internal: edgeDetector */
57    /* Transition: '<S1>:37' */
58    /* Transition: '<S1>:38' */
59    if (edgeDetector_U.edgeDetector_in == 1) {
60      /* Outport: '<Root>/edgeDetector_out' */
61      /* Transition: '<S1>:9' */
62      /* Transition: '<S1>:10' */
63      edgeDetector_Y.edgeDetector_out = 0;
64      edgeDetector_DW.is_c3_edgeDetector = edgeDetector_IN_S1;
65    } else {
66      /* Outport: '<Root>/edgeDetector_out' */
67      /* Transition: '<S1>:7' */
68      /* Transition: '<S1>:23' */
69      edgeDetector_Y.edgeDetector_out = 0;
70      edgeDetector_DW.is_c3_edgeDetector = edgeDetector_IN_S0;
71    }
72  } else {
73    switch (edgeDetector_DW.is_c3_edgeDetector) {
74     case edgeDetector_IN_S0:
75      /* During 'S0': '<S1>:20' */
76      if (edgeDetector_U.edgeDetector_in == 1) {
77        /* Outport: '<Root>/edgeDetector_out' */
78        /* Transition: '<S1>:26' */
79        /* Transition: '<S1>:16' */
80        edgeDetector_Y.edgeDetector_out = 1;
81        edgeDetector_DW.is_c3_edgeDetector = edgeDetector_IN_S1Edge;
82      }
83      break;
84
85     case edgeDetector_IN_S0Edge:
86      /* During 'S0Edge': '<S1>:3' */
87      if (edgeDetector_U.edgeDetector_in == 1) {
88        /* Outport: '<Root>/edgeDetector_out' */
89        /* Transition: '<S1>:15' */
90        /* Transition: '<S1>:25' */
91        /* Transition: '<S1>:16' */
92        edgeDetector_Y.edgeDetector_out = 1;
93        edgeDetector_DW.is_c3_edgeDetector = edgeDetector_IN_S1Edge;
94      } else {
```

```
95          /* Outport: '<Root>/edgeDetector_out' */
96          /* Transition: '<S1>:22' */
97          /* Transition: '<S1>:23' */
98          edgeDetector_Y.edgeDetector_out = 0;
99          edgeDetector_DW.is_c3_edgeDetector = edgeDetector_IN_S0;
100       }
101       break;
102
103     case edgeDetector_IN_S1:
104       /* During 'S1': '<S1>:27' */
105       if (edgeDetector_U.edgeDetector_in == 0) {
106         /* Outport: '<Root>/edgeDetector_out' */
107         /* Transition: '<S1>:33' */
108         /* Transition: '<S1>:13' */
109         edgeDetector_Y.edgeDetector_out = 1;
110         edgeDetector_DW.is_c3_edgeDetector = edgeDetector_IN_S0Edge;
111       }
112       break;
113
114     default:
115       /* During 'S1Edge': '<S1>:4' */
116       if (edgeDetector_U.edgeDetector_in == 0) {
117         /* Outport: '<Root>/edgeDetector_out' */
118         /* Transition: '<S1>:12' */
119         /* Transition: '<S1>:32' */
120         /* Transition: '<S1>:13' */
121         edgeDetector_Y.edgeDetector_out = 1;
122         edgeDetector_DW.is_c3_edgeDetector = edgeDetector_IN_S0Edge;
123       } else {
124         /* Outport: '<Root>/edgeDetector_out' */
125         /* Transition: '<S1>:29' */
126         /* Transition: '<S1>:30' */
127         edgeDetector_Y.edgeDetector_out = 0;
128         edgeDetector_DW.is_c3_edgeDetector = edgeDetector_IN_S1;
129       }
130       break;
131   }
132 }
133
134 /* End of Chart: '<Root>/edgeDetector' */
135
136 /* Matfile logging */
137 rt_UpdateTXYLogVars(edgeDetector_M->rtwLogInfo,
138                     (&edgeDetector_M->Timing.taskTime0));
139
140 /* signal main to stop simulation */
141 {                                    /* Sample time: [0.01s, 0.0s] */
142   if ((rtmGetTFinal(edgeDetector_M)!=-1) &&
```

```
143         !((rtmGetTFinal(edgeDetector_M)-edgeDetector_M->Timing.taskTime0
    ) >
144           edgeDetector_M->Timing.taskTime0 * (DBL_EPSILON))) {
145       rtmSetErrorStatus(edgeDetector_M, "Simulation finished");
146     }
147   }
148
149   /* Update absolute time for base rate */
150   /* The "clockTick0" counts the number of times the code of this task
     has
151    * been executed. The absolute time is the multiplication of "
     clockTick0"
152    * and "Timing.stepSize0". Size of "clockTick0" ensures timer will not
153    * overflow during the application lifespan selected.
154    * Timer of this task consists of two 32 bit unsigned integers.
155    * The two integers represent the low bits Timing.clockTick0 and the
     high bits
156    * Timing.clockTickH0. When the low bit overflows to 0, the high bits
     increment.
157    */
158   if (!(++edgeDetector_M->Timing.clockTick0)) {
159     ++edgeDetector_M->Timing.clockTickH0;
160   }
161
162   edgeDetector_M->Timing.taskTime0 = edgeDetector_M->Timing.clockTick0 *
163     edgeDetector_M->Timing.stepSize0 + edgeDetector_M->Timing.
     clockTickH0 *
164     edgeDetector_M->Timing.stepSize0 * 4294967296.0;
165 }
166
167 /* Model initialize function */
168 void edgeDetector_initialize(void)
169 {
170   /* Registration code */
171
172   /* initialize non-finites */
173   rt_InitInfAndNaN(sizeof(real_T));
174
175   /* initialize real-time model */
176   (void) memset((void *)edgeDetector_M, 0,
177                 sizeof(RT_MODEL_edgeDetector_T));
178   rtmSetTFinal(edgeDetector_M, -1);
179   edgeDetector_M->Timing.stepSize0 = 0.01;
180
181   /* Setup for data logging */
182   {
183     static RTWLogInfo rt_DataLoggingInfo;
184     rt_DataLoggingInfo.loggingInterval = NULL;
```

```
185     edgeDetector_M->rtwLogInfo = &rt_DataLoggingInfo;
186   }
187
188   /* Setup for data logging */
189   {
190     rtliSetLogXSignalInfo(edgeDetector_M->rtwLogInfo, (NULL));
191     rtliSetLogXSignalPtrs(edgeDetector_M->rtwLogInfo, (NULL));
192     rtliSetLogT(edgeDetector_M->rtwLogInfo, "tout");
193     rtliSetLogX(edgeDetector_M->rtwLogInfo, "");
194     rtliSetLogXFinal(edgeDetector_M->rtwLogInfo, "");
195     rtliSetLogVarNameModifier(edgeDetector_M->rtwLogInfo, "rt_");
196     rtliSetLogFormat(edgeDetector_M->rtwLogInfo, 4);
197     rtliSetLogMaxRows(edgeDetector_M->rtwLogInfo, 0);
198     rtliSetLogDecimation(edgeDetector_M->rtwLogInfo, 1);
199     rtliSetLogY(edgeDetector_M->rtwLogInfo, "");
200     rtliSetLogYSignalInfo(edgeDetector_M->rtwLogInfo, (NULL));
201     rtliSetLogYSignalPtrs(edgeDetector_M->rtwLogInfo, (NULL));
202   }
203
204   /* states (dwork) */
205   (void) memset((void *)&edgeDetector_DW, 0,
206                 sizeof(DW_edgeDetector_T));
207
208   /* external inputs */
209   edgeDetector_U.edgeDetector_in = 0;
210
211   /* external outputs */
212   edgeDetector_Y.edgeDetector_out = 0;
213
214   /* Matfile logging */
215   rt_StartDataLoggingWithStartTime(edgeDetector_M->rtwLogInfo, 0.0,
       rtmGetTFinal
216     (edgeDetector_M), edgeDetector_M->Timing.stepSize0, (&
       rtmGetErrorStatus
217     (edgeDetector_M)));
218
219   /* SystemInitialize for Chart: '<Root>/edgeDetector' */
220   edgeDetector_DW.is_active_c3_edgeDetector = 0U;
221   edgeDetector_DW.is_c3_edgeDetector = edgeDetector_IN_NO_ACTIVE_CHILD;
222
223   /* SystemInitialize for Outport: '<Root>/edgeDetector_out'
       incorporates:
224    *  SystemInitialize for Chart: '<Root>/edgeDetector'
225    */
226   edgeDetector_Y.edgeDetector_out = 0;
227 }
228
229 /* Model terminate function */
```

```
230 void edgeDetector_terminate(void)
231 {
232   /* (no terminate code required) */
233 }
```

Listing B.4: *Edge Detector - EdgeDetector.c*

Appendix C

Stateflow Verification Code

In this part, the complete code to run the *Stateflow* verification is shown. An example model is shown in Figure C.1. The coverage part of the test requires a "testData" file, which is structured as depicted in Listing C.1. Afterward the code for all necessary files is listed.

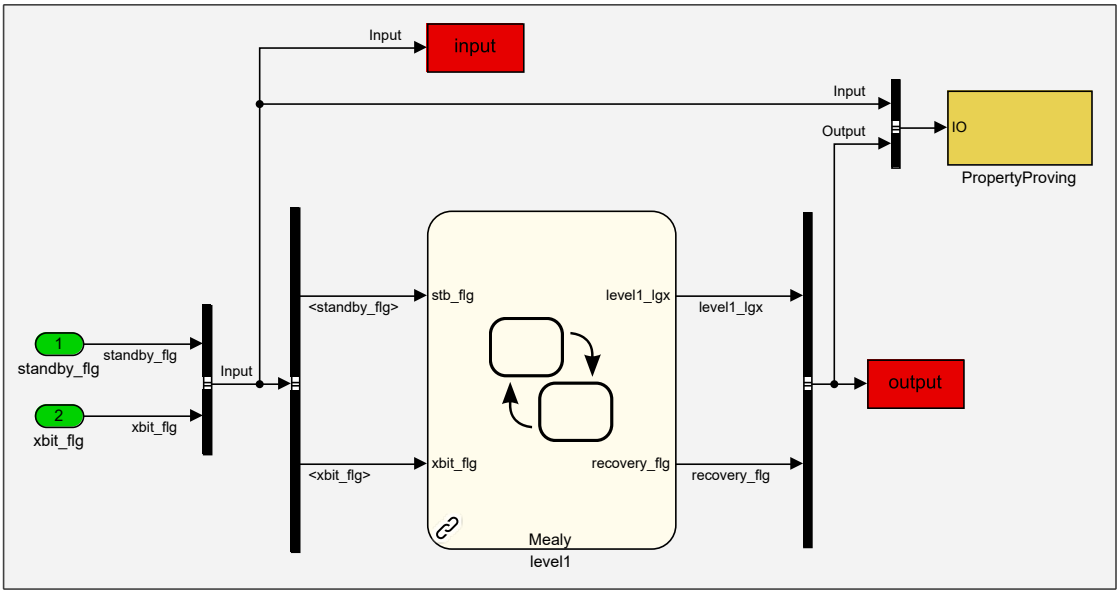


Figure C.1: *Example Simulink Model*

```
testData
  1x3 struct array with fields:
    name
    input

testData(1).name
  'Zero Input'

testData(1).input
  struct with fields:
    standby_flg: [1x timeseries]
    xbit_flg: [1x timeseries]
```

Listing C.1: *Stateflow Verification - TestData.mat*

```
close all;
clear all;
clc;

addpath(pwd);
run('InitModel.m');  % Call all functions necessary to run models

% Single Run
modelName = 'level1';

InitTestRun; % Init test run and save temporary workspace file
DesignErrorDetection_DeadLogic;
DesignErrorDetection_IoDz;
PropertyProving;
Coverage;

delete('tempWorkspace.mat') % Clear workspace file
```

Listing C.2: *Stateflow Verification - RunSingle.m*

```
close all;
clear all;
clc;

addpath(pwd);
run('InitTest.m');

model_names = { 'level1',
                'level2_opl',
                'level3_opl_ep',
                'level3_opl_epll',
                'level3_opl_fo',
                'level3_opl_foll',
                'level4_opl_fo_rtb',
                'level3_opl_fo_loiter',
};

modelNameI = 0;
save('tempWorkspace.mat');

while modelNameI < length(model_names)
    modelNameI = modelNameI + 1;
    modelName = char(model_names(modelNameI));
    save('tempWorkspace.mat');

    InitTestRun;
        DesignErrorDetection_DeadLogic;
        DesignErrorDetection_IoDz;
        PropertyProving;
        Coverage;
  load('tempWorkspace.mat');
end

delete('tempWorkspace.mat')
```

Listing C.3: *Stateflow Verification - RunAll.m*

```
fprintf('Initializing Coverage, Design Error Detection, Property Proving
    \n');
fprintf('\tModel Name: %s\n', modelName);
save('tempWorkspace.mat')

if isfolder(modelName)
    rmdir(modelName,'s')
end
mkdir(modelName);
cd(modelName);
open(modelName);

sflow=sfroot;
myCharts = sflow.find('-isa', 'Stateflow.Chart', '-and', 'Name',
    modelName);
if myCharts.length == 1
    sfprint(myCharts, 'pdf', 'stateflow');
else
    warning('Wrong Number of Stateflow Charts');
end

myBoxes = sflow.find('-isa', 'Stateflow.Box', '-and', 'Name', 'tc');
if myBoxes.length >= 1
    for i=1:length(myBoxes)
        if strcmp(extractAfter(myBoxes(i).Path, '/'), modelName)
            sfprint(myBoxes(i), 'pdf', 'tc');
        end
    end
else
    warning('No Stateflow Box named tc');
end

myBoxes = sflow.find('-isa', 'Stateflow.Box', '-and', 'Name', 'ta');
if myBoxes.length >= 1
    for i=1:length(myBoxes)
        if strcmp(extractAfter(myBoxes(i).Path, '/'), modelName)
            sfprint(myBoxes(i), 'pdf', 'ta');
        end
    end
else
    warning('No Stateflow Box named ta')
end

fprintf('\t\tFinished\n');
bdclose(modelName);
cd ..
```

Listing C.4: *Stateflow Verification - InitTestRun.m*

```
function  InitSLDVTest(modelName, folderName)
    if isfile('tempWorkspace.mat')
        clearvars -except modelName folderName
        load('tempWorkspace.mat')
    end
    folderName=['temp_' modelName folderName];
    if isfolder(folderName)
        rmdir(folderName,'s')
    end
    mkdir(folderName);
    cd(folderName);
    open(modelName);
end
```

Listing C.5: *Stateflow Verification - InitSLDVTest.m*

```
function  CloseSLDVTest(modelName, folderName)
    folderName=['temp_' modelName folderName];
    fprintf('\t\tFinished\n');
    bdclose(modelName);
    bdclose('all');
    cd ..
    rmdir(folderName, 's');
    if isfile('tempWorkspace.mat')
        clear all;
    end
end
```

Listing C.6: *Stateflow Verification - CloseSLDVTest.m*

```
fprintf('\tRunning Design Error Detection - Dead Logic Test\n');
InitSLDVTest(modelName, '_designErrorDetection_deadLogic');
% - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

opts = sldvoptions;
opts.DetectDeadLogic = 'on';
opts.DetectDivisionByZero = 'off';
opts.DetectIntegerOverflow = 'off';
opts.DetectOutOfBounds = 'off';
opts.DisplayReport = 'off';
opts.Mode = 'DesignErrorDetection';
%opts.ModelCoverageObjectives = 'MCDC';
opts.outputDir = 'sldv_output/'; %opts.outputDir = 'sldv_output/
    $ModelName$';
opts.ReportFileName = '$ModelName$_ded_dl';
opts.ReportIncludeGraphics = 'on';
opts.ReportPDFFormat = 'on';
opts.SaveHarnessModel = 'on';
opts.SaveReport = 'on';

[status, fileNames] = sldvrun(modelName, opts, true);

% - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
copyfile(['sldv_output/' modelName '_ded_dl.pdf'], ['../' modelName '/
    ded_dl.pdf'])
CloseSLDVTest(modelName, '_designErrorDetection_deadLogic');
```

Listing C.7: *Stateflow Verification - DesignErrorDetection_DeadLogic.m*

```
fprintf('\tRunning Design Error Detection - Interger Overflow, Devision
    by Zero Test\n');
InitSLDVTest(modelName, '_designErrorDetection_IoDz');
% - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

opts = sldvoptions;
opts.DetectDeadLogic = 'off';
opts.DetectDivisionByZero = 'on';
opts.DetectIntegerOverflow = 'on';
opts.DetectOutOfBounds = 'on';
opts.DisplayReport = 'off';
opts.Mode = 'DesignErrorDetection';
%opts.ModelCoverageObjectives = 'MCDC';
opts.outputDir = 'sldv_output/'; %opts.outputDir = 'sldv_output/
    $ModelName$';
opts.ReportFileName = '$ModelName$_ded_iodz';
opts.ReportIncludeGraphics = 'on';
opts.ReportPDFFormat = 'on';
opts.SaveHarnessModel = 'on';
opts.SaveReport = 'on';

[status, fileNames] = sldvrun(modelName, opts, true);

% - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
copyfile(['sldv_output/' modelName '_ded_iodz.pdf'], ['../' modelName '/
    ded_iodz.pdf'])
CloseSLDVTest(modelName, '_designErrorDetection_IoDz');
```

Listing C.8: *Stateflow Verification - DesignErrorDetection_IoDz.m*

```
fprintf('\tRunning Property Proving Test\n');
InitSLDVTest(modelName, '_propertyProving');
% - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

propertyProvingExists = 1;

try
    set_param([modelName '/PropertyProving'],'commented','off');
catch exception
    propertyProvingExists = 0;
    fprintf('\t\tNo Property Proving\n');
    warning('Property Proving does not exist');
end

if propertyProvingExists

    opts = sldvoptions;
    %opts.DetectDeadLogic = 'off';
    %opts.DetectDivisionByZero = 'on';
    %opts.DetectIntegerOverflow = 'on';
    %opts.DetectOutOfBounds = 'on';
    opts.DisplayReport = 'off';
    opts.Mode = 'PropertyProving';
    %opts.ModelCoverageObjectives = 'MCDC';
    opts.outputDir = 'sldv_output/'; %opts.outputDir = 'sldv_output/
    $ModelName$';
    opts.ReportFileName = '$ModelName$_pp';
    opts.ReportIncludeGraphics = 'on';
    opts.ReportPDFFormat = 'on';
    opts.SaveHarnessModel = 'on';
    opts.SaveReport = 'on';

    [status, fileNames] = sldvrun(modelName, opts, true);
    %set_param([modelName '/PropertyProving'],'commented','on');
end

% - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
if propertyProvingExists
    copyfile(['sldv_output/' modelName '_pp.pdf'], ['../' modelName '/
    pp.pdf'])
end
CloseSLDVTest(modelName, '_propertyProving');
```

Listing C.9: *Stateflow Verification - PropertyProving.m*

```
fprintf('\tRunning Covarage Test\n');
InitSLDVTest(modelName, '_coverage');
% - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
load(['testData_' modelName '.mat']);
numberOfTests = length(testData);

% Set model parameter
set_param(modelName, 'RecordCoverage', 'on', 'CovModelRefEnable', 'all',
    'CovMetricSettings', 'dcmtrzoidwe');
set_param(modelName, 'LoadExternalInput', 'on', 'ExternalInput',
   Coverage_InputString(testData(1).input));

fprintf('\t\tTotal Coverage Tests: %d\n', numberOfTests);
for i=1:1:numberOfTests
    fprintf('\t\t\tRunning Test %d/%d - %s\n', i, numberOfTests,
    testData(i).name);
    input = testData(i).input; % Load input data

    % Choosing normal sim run to be compatible with design verifier
    fields = fieldnames(input);
    %Readout end time from one input field and set stop time in Simulink
    set_param(modelName,'StopTime', num2str(input.(fields{1})
    .TimeInfo.End));
    sim(modelName);
    cvdos{i} = covdata;

    Coverage_plotData(modelName, testData(i).name, i, 'input', input);
    Coverage_plotData(modelName, testData(i).name, i, 'output', output);
end
% Create cumulative report for all tests
cvhtml(modelName, cvdos{:}, '-sRT=0');

% - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Coverage_copyFiles(modelName);
CloseSLDVTest(modelName, '_coverage');
```

Listing C.10: *Stateflow Verification - Coverage.m*

```
function stTemp = Coverage_InputString(input)

fields = fieldnames(input);

% Generate string of input variables
stTemp = '';
for i=1:1:length(fields)
    stTemp = [stTemp 'input.' fields{i}];

    if (i< length(fields))
        stTemp = [stTemp ', '];
    end
end

end
```

Listing C.11: *Stateflow Verification - Coverage_InputString.m*

```
function Coverage_plotData(modelName, testName, numTest, stDir, data)

    st_Name = [modelName ' - Test' sprintf('%02d', numTest) ' - ' stDir
    ' - ' testName];
    h = figure('NumberTitle', 'off', 'Name', st_Name);
    number_of_inputs = size(fieldnames(data));
    number_of_inputs = number_of_inputs(1);

    in_fields = fieldnames(data);

    for k=1:1:number_of_inputs
        subplot(number_of_inputs, 1, k);
        plot(data.(in_fields{k}))
        grid on;
        title('');

        if (k < number_of_inputs)
            xlabel('');
        end
    end
    savefig(h, st_Name);
    close(h);

end
```

Listing C.12: *Stateflow Verification - Coverage_PlotData.m*

```
function Coverage_copyFiles(modelName)

    filenames=dir('*.html');
    for i=1:length(filenames)
        copyfile(filenames(i).name, ['../' modelName '/' filenames(i)
    .name])
    end

    filenames=dir('*.fig');
    for i=1:length(filenames)
        copyfile(filenames(i).name, ['../' modelName '/' filenames(i)
    .name])
    end

    filenames=dir('scv_images/*.*');
    filenames = filenames(~ismember({filenames.name}, {'.', '..'})); %
    remove '.' and '..'
    mkdir(['../' modelName '/scv_images/']);
    for i=1:length(filenames)
        copyfile(['scv_images\' filenames(i).name], ['../' modelName '/
    scv_images/' filenames(i).name]);
    end

end
```

Listing C.13: *Stateflow Verification - Coverage_CopyFiles.m*

Appendix D

FCSA Transition Conditions/Actions

In the following, all transition condition and transition action subsystems of the *Operator-Centric Multi-User Flight Control System Automation for experimental UAVs and OPVs with Contingency Procedures (FCSA)* are depicted.

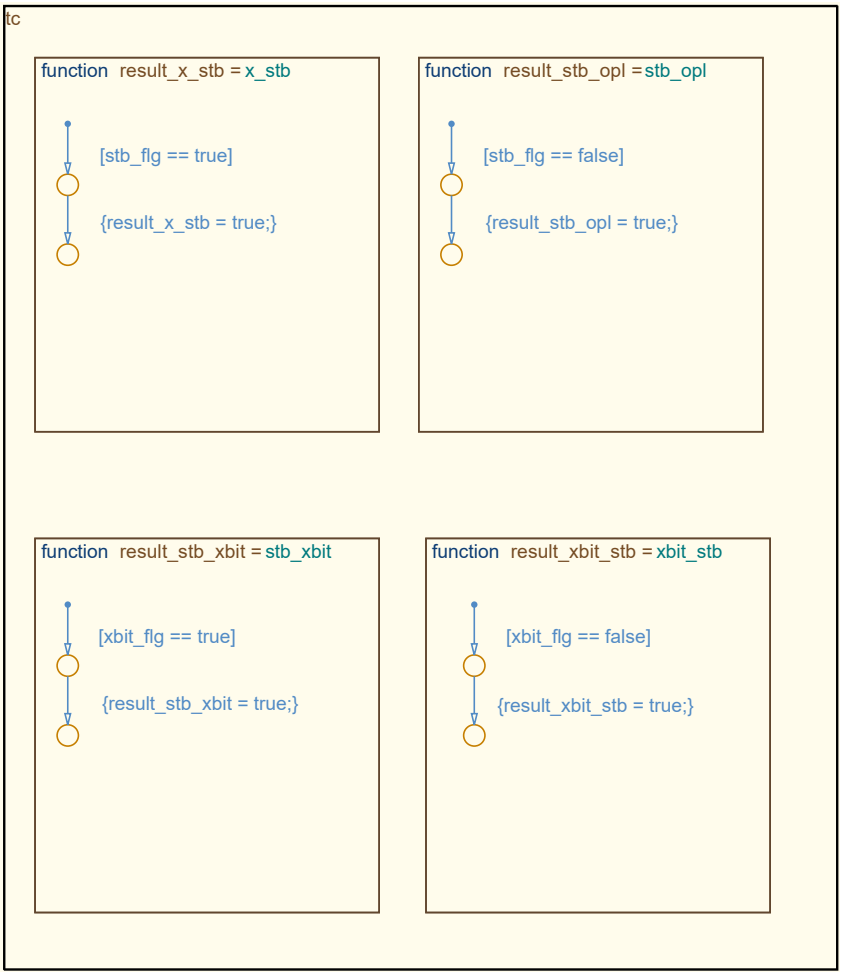


Figure D.1: *State Machine Level 1 - Transition Conditions*

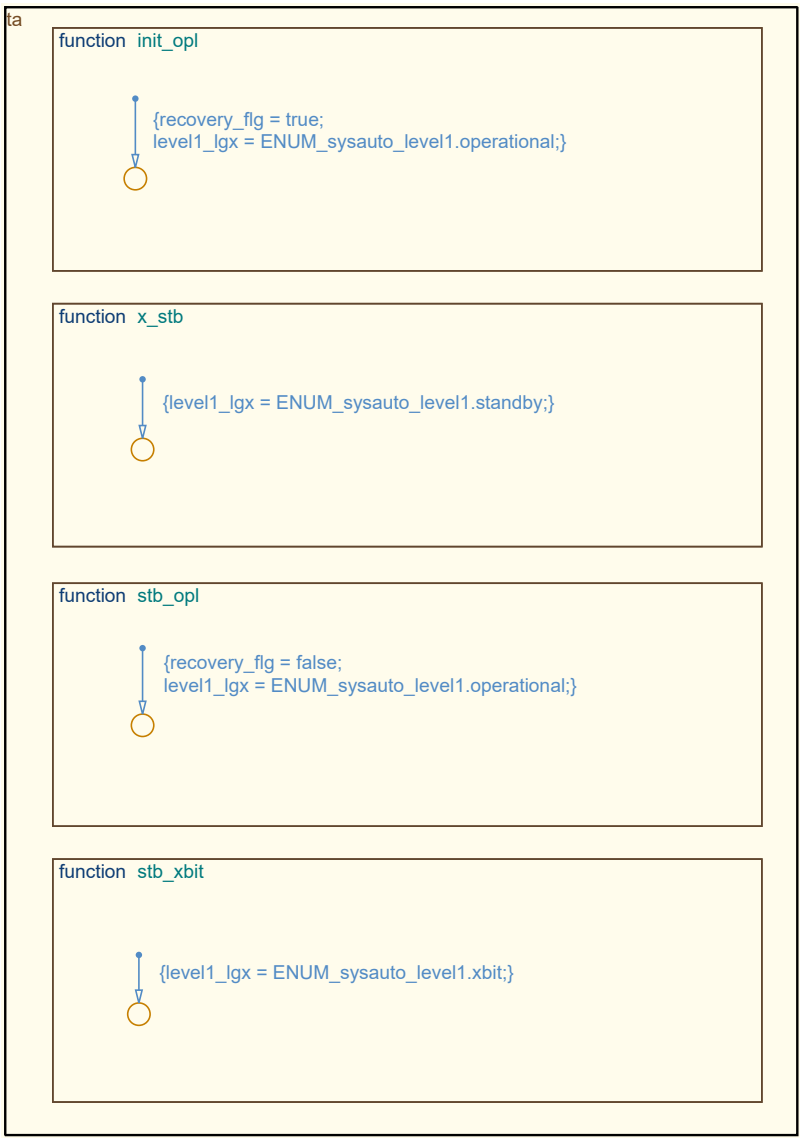


Figure D.2: *State Machine Level 1 - Transition Actions*

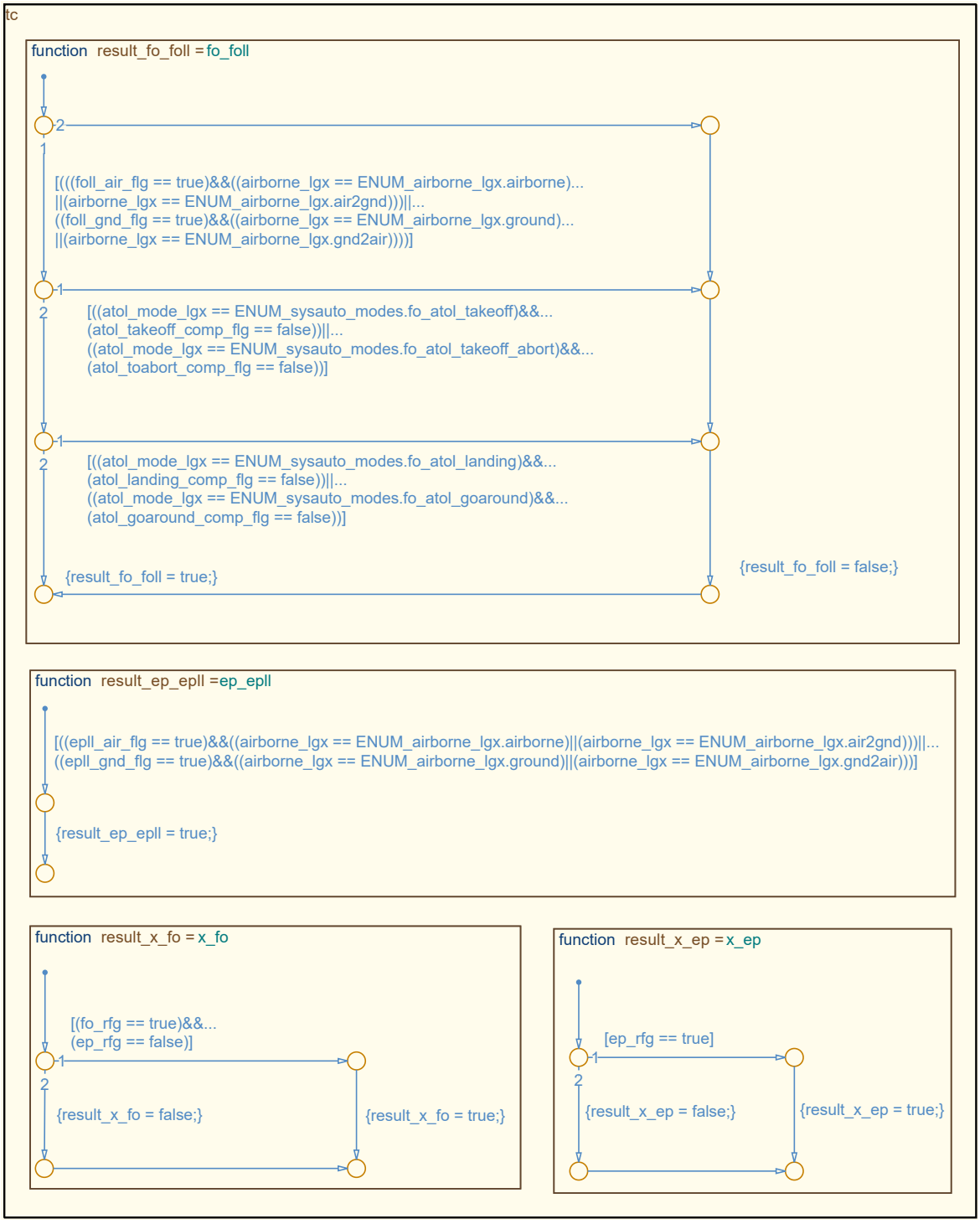


Figure D.3: *State Machine Level 2 - Transition Conditions*

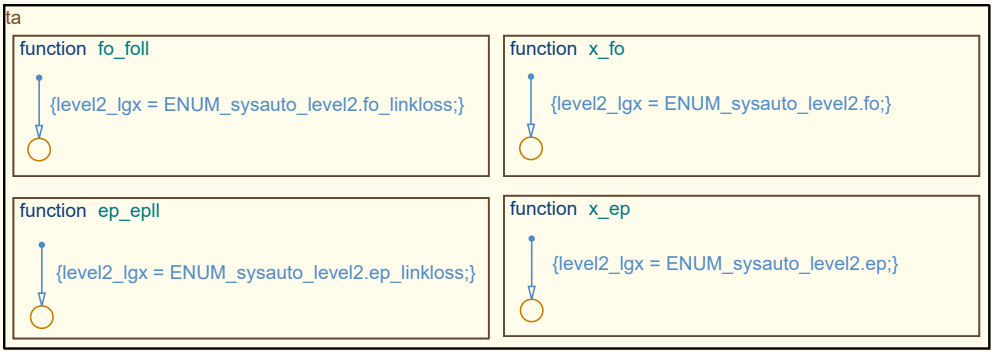


Figure D.4: *State Machine Level 2 - Transition Actions*

Figure D.5: *State Machine Level 3 - EP - Transition Conditions*

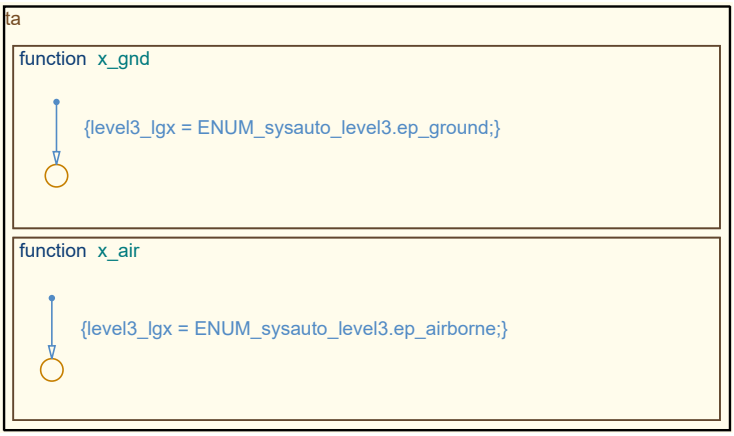


Figure D.6: *State Machine Level 3 - EP - Transition Actions*

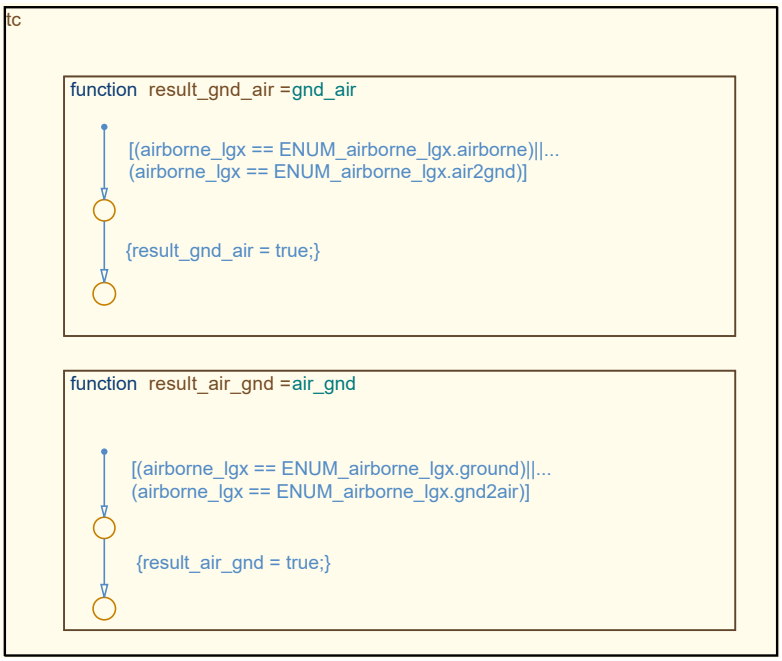


Figure D.7: *State Machine Level 3 - EPLL - Transition Conditions*

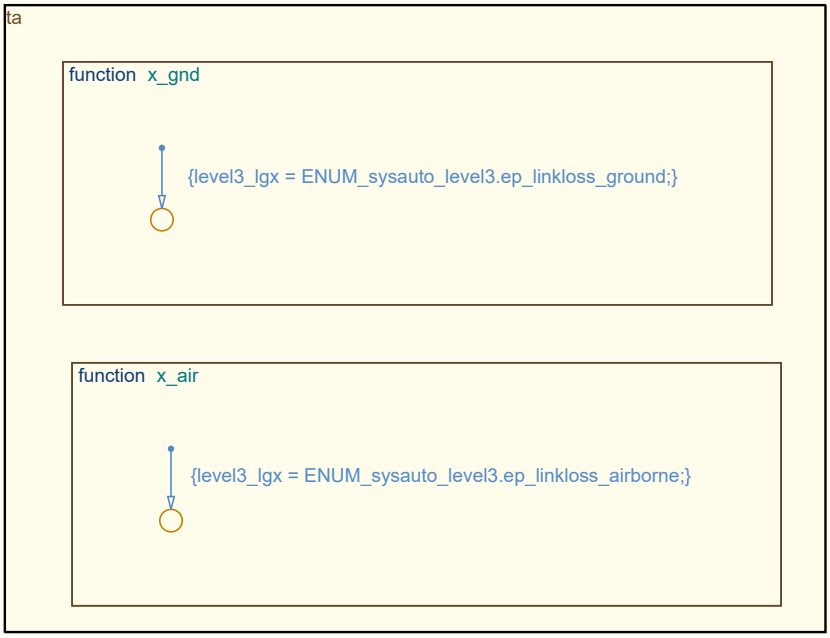


Figure D.8: *State Machine Level 3 - EPLL - Transition Actions*

Figure D.9: *State Machine Level 3 - FO - Transition Conditions*

Figure D.10: *State Machine Level 3 - FO - Transition Actions*

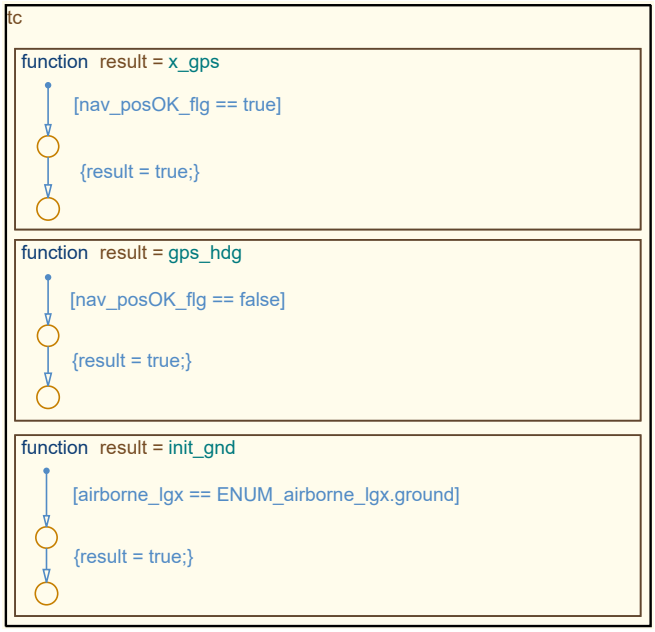


Figure D.11: *State Machine Level 3 - FOLL - Transition Conditions*

Figure D.12: *State Machine Level 3 - FOLL - Transition Actions*

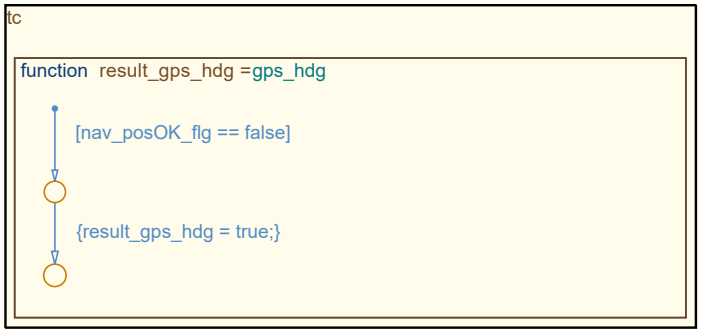


Figure D.13: *State Machine Level 4 - Transition Conditions*

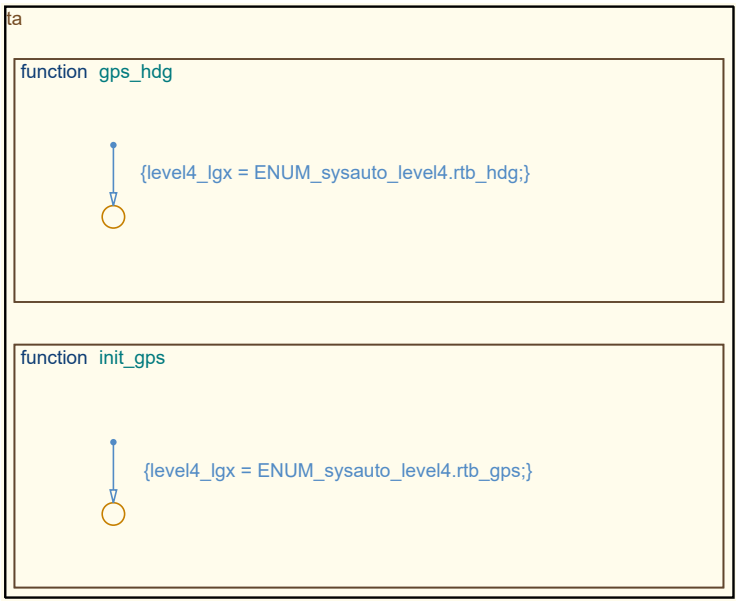


Figure D.14: *State Machine Level 4 - Transition Actions*

Figure D.15: *State Machine Level 3 - Loiter - Transition Conditions*

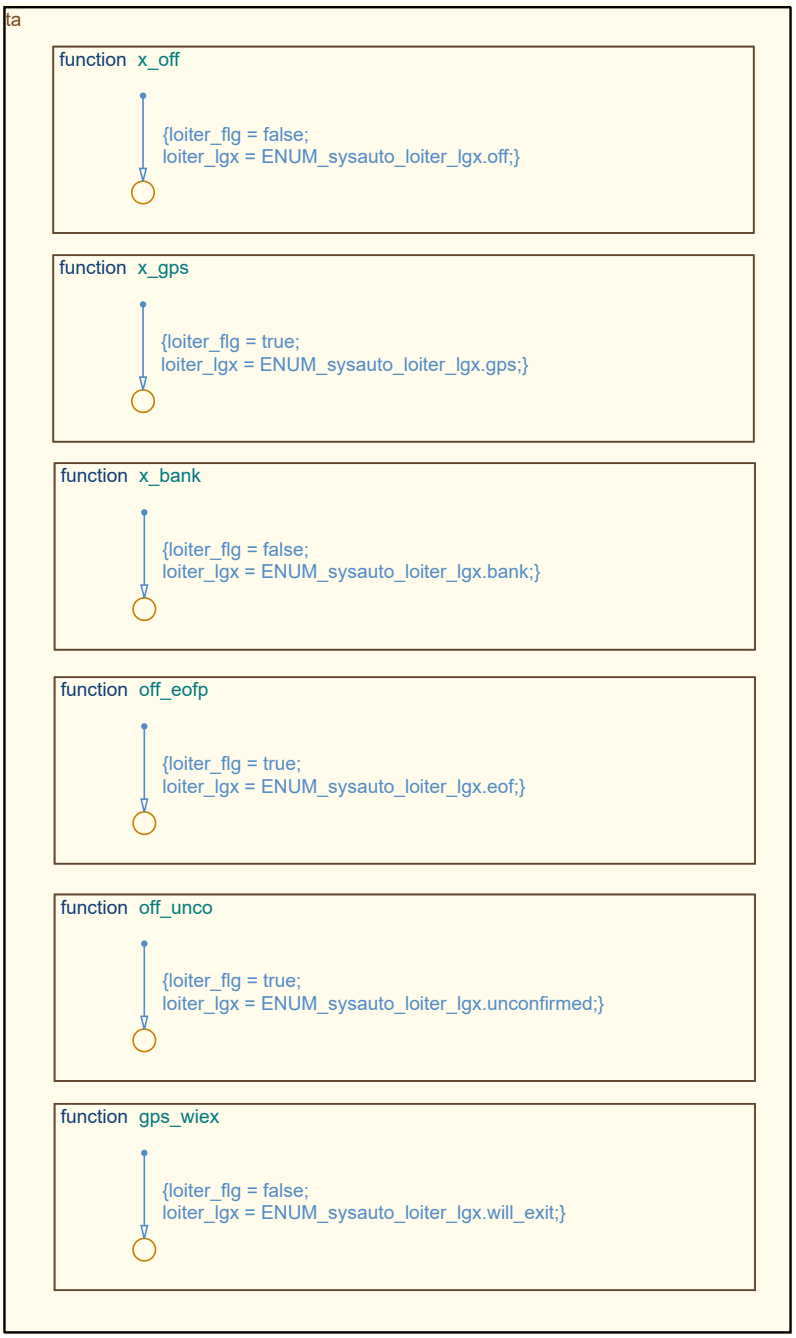


Figure D.16: *State Machine Level 3 - Loiter - Transition Actions*

Appendix E

FCSA Unit Tests

In Table E.1, all unit tests of the *Operator-Centric Multi-User Flight Control System Automation for experimental UAVs and OPVs with Contingency Procedures (FCSA)* are listed with their test number and name.

Table E.1: *FCSA - Unit Tests*

Number	**Name**
3001	Zero
3002	Normal Flight
3003	Takeoff Abort
3004	EP Link Loss Ground
3005	FO/EP in Loop combinations
3006	Standby/Experimental Build-in-Test combinations
3007	EP Takeoff
3008	EP Link Loss Airborne
3009	FO Link Loss Airborne
3010	MLC HLC Superposition combinations
3011	FO HLC Loiter WP
3012	HLC immediate Loiter
3013	FO RTB

Appendix F

FTMI Transition Conditions/Actions

In the following, all transition condition and transition action subsystems of the *Multi-Maneuver Multi-Control-Level Flight Test Maneuver Injection with automatic Trim Point Capture and Verification (FTMI)* are depicted.

Figure F.1: *State Machine Level 1 - Transition Conditions*

Figure F.2: *State Machine Level 1 - Transition Actions*

Bibliography

[Acc1997] Accident Investigation Division - Civil Aviation Department - Hong Kong: Aircraft Accident Report - Report on the Incident to Airbus A320-231 VR-HYU at Hong Kong International Airport on 6 June 1994. (1997). https://aviation-safety.net/reports/1994/19940606_A320_VR-HYU.pdf. [Accessed: 21.02.2020]

[Ace2016] Acentiss: elias - The future of electrically-powered flight. (2016). http://www.acentiss.de/system/files_force/downloads/folder_acentiss_elias_1.pdf. [Accessed: 04.04.2019]

[AEC2011] AECC: *ARINC Specification 424-20 — Navigation System Database*. 2011

[Air1987a] Airbus: Airbus A320 - Flight Crew Operating Manual - Auto Flight. (1987). https://www.smartcockpit.com/aircraft-ressources/A320-Autoflight.html. [Accessed: 21.02.2020]

[Air1987b] Airbus: Airbus A320 - Flight Crew Operating Manual - Flight Controls. (1987). https://www.smartcockpit.com/aircraft-ressources/A320-Flight_Controls.html. [Accessed: 21.02.2020]

[Air2017] Airbus: Successful first flight for UAV demonstrator SAGITTA. (2017). https://www.airbus.com/newsroom/press-releases/en/2017/07/successful-first-flight-for-uav-demonstrator-sagitta.html. [Accessed: 09.04.2019]

[Air1996] Aircraft Accident Investigation Commission - Ministry of Transport: Aircraft Accident Investigation Report - China Airlines - Airbus Industrie A300B4-622R, B1816 - Nagoya Airport - April 26, 1994. (1996). https://ocw.mit.edu/ans7870/16/16.63j/f12/MIT16_63JF12_A300.pdf. [Accessed: 21.02.2020]

[Alb1991] Albers, James A.: Aviation Safety and Automation Technology for Subsonic Transports. In: *NASA Technical Memorandum 103831*, 1991, p. 1–54

[Amb2016] Ambrose, Kevin: Reflecting Pool takeoff, buzzing the Capitol and the 100th anniversary of the 'mile-high club'. (2016). https://www.washingtonpost.com/news/capital-weather-gang/wp/2016/08/30/reflecting-pool-takeoff-buzzing-the-capitol-and-the-100th-anniversary-of-the-mile-high-club/. [Accessed: 11.09.2019]

[Ame2017] Ameri Research Inc.: Global Unmanned Aerial Vehicle (UAV) Market To 2024. (2017). https://www.ameriresearch.com/product/unmanned-aerial-vehicle-uav-market/. [Accessed: 31.05.2020]

[ADN1992] Ashar, Pranav; Devadas, Srinivas; Newton, A. R.: Finite State Machine Decomposition. In: *Sequential Logic Synthesis*, 1992, p. 117–168

[Aus2010] Austin, Reg: *Introduction to Unmanned Aircraft Systems (UAS)*. John Wiley & Sons, Ltd, 2010. – ISBN 9780470664797

[Bai1983] Bainbridge, Lisanne: Ironies of automation. In: *Automatica*, 1983, p. 775–779

[BCCZ1999] Biere, Armin; Cimatti, Alessandro; Clarke, Edmund; Zhu, Yunshan: Symbolic Model Checking without BDDs. In: *Proc. of Fifth Int. Conf. on Tools for Construction and Analysis of Systems, TACAS '99*, 1999, p. 1–15

[Bil1991] Billings, Charles E.: Human-centered aircraft automation: A concept and guidelines. In: *NASA Technical Memorandum 103885*, 1991, p. 3–120

[Bil1996] Billings, Charles E.: Human-Centered Aviation Automation: Principles and Guidelines. In: *NASA Technical Memorandum 110381*, 1996, p. 3–219

[Bil1997] Billings, Charles E.: *Aviation Automation: The Search for a Human-Centered Approach*. CRC Press LLC, 1997. – ISBN 9781351464932

[Boe2002] Boeing: X-45 Joint Unmanned Combat Air System. (2002). https://www.boeing.com/history/products/x-45-joint-unmanned-combat-air-system.page. [Accessed: 10.02.2020]

[Boo1847] Boole, George: *The Mathematical Analysis of Logic – Being an Essay Towards a Calculus of Deductive Reasoning*. 1847

[Boo1854] Boole, George: *An Investigation of The Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*. 1854

[BGHH2014] Braun, Stanislav; Geiser, Markus; Heller, Matthias; Holzapfel, Florian: Configuration Assessment and Preliminary Control Law Design for a Novel Diamond-Shaped UAV. In: *International Conference on Unmanned Aircraft Systems (ICUAS)*, 2014, p. 1009–1020

[BWB+2012] Braun, Stanislav; Wolze, Philip; Braun, Benjamin; Geiser, Markus; Holzapfel, Florian: Hardware-in-the-Loop Simulation in the Context of the Development of Automatic Flight Control Systems for UAV. In: *AIRTEC UAV World*, 2012, p. 1–8

[Bur1989] Bureau of Enquiry and Analysis for Civil Aviation Safety: Final Report concerning the accident which occurred on June 26th 1988 at Mulhouse-Habsheim to the Airbus A320, registered F-GFKC. (1989). https://reports.aviation-safety.net/1988/19880626-0_A320_F-GFKC.pdf. [Accessed: 21.02.2020]

[Bur2012] Bureau of Enquiry and Analysis for Civil Aviation Safety: Final Report on the accident on 1st June 2009 to the Airbus A330-203 registered F-GZCP operated by Air France flight AF 447 Rio de Janerio - Paris. (2012). https://www.smartcockpit.com/docs/airbus-a330-air-france-aerodynamics-stall--crash.pdf. [Accessed: 21.02.2020]

[CGP1999] Clarke, Edmund; Grumberg, Orna; Peled, Doron: *Model Checking*. MIT Press, 1999. – ISBN 9780262032704

[Cla1997] Clarke, Edmund. M.: Model Checking. In: *Foundations of Software Technology and Theoretical Computer Science*, 1997, p. 54–56

[Con1971] Conway, John H.: *Regular Algebra and Finite Machines*. Dover Publications, Inc., 1971. – ISBN 0412106205

[Cou2015] Courtland, Rachel: Gordon Moore: The Man Whose Name Means Progress. (2015). https://spectrum.ieee.org/computing/hardware/gordon-moore-the-man-whose-name-means-progress. [Accessed: 19.09.2019]

[Cur1985] Curry, Renwick E.: The Introduction of New Cockpit Technology: A Human Factors Study. In: *NASA Technical Memorandum 86659*, 1985, p. 1–68

[DLR2013] Dalldorff, Lothar; Luckner, Robert; Reichel, Reinhard: A Full-Authority Automatic Flight Control System for the Civil Airborne Utility Aircraft S 15 – LAPAZ. In: *CEAS EuroGNC*, 2013, p. 1–20

[Das2003] Dassault Aviation: Dassault nEUROn - Introduction. (2003). https://www.dassault-aviation.com/en/defense/neuron/introduction/. [Accessed: 10.02.2020]

[Dia2012] Diamond Aircraft: Airplane Flight Manual - DA 42 NG. (2012). http://support.diamond-air.at/fileadmin/uploads/files/after_sales_support/DA42_New_Generation/Airplane_Flight_Manual_with_MAM42-600_DA42-VI/Basic_Manual/70116e-r4-complete.pdf. [Accessed: 11.02.2021]

[Dia2018] Diamond Aircraft: DA42-VI THE DEFINITION OF PERFECTION. (2018). https://www.diamondaircraft.com/fileadmin/diamondaircraft/documents/da42/DA42-VI_Product_Folder_20192503_SCREEN.pdf. [Accessed: 09.04.2019]

[DLR2018] DLR: Dornier Do 228-101 D-CODE. (2018). https://www.dlr.de/content/en/articles/aeronautics/research-fleet-infrastructure/dlr-research-aircraft/dornier-do-228-101-d-code.html. [Accessed: 30.10.2019]

[DLR2019] DLR: Dornier 228-101 (D-CODE). (2019). https://www.dlr.de/content/de/grossforschungsanlagen/dornier-228-101.html. [Accessed: 30.10.2019]

[Dri2016] Dries, C.: Automatischer Start und Landung einer Diamond DA42, Technologien der Universität Stuttgart. (2016). http://www.dglr.de/publikationen/2016/010002.pdf. [Accessed: 30.11.2019]

[DHB+1988] Duke, Eugene L.; Hewett, Marle; Brumbaugh, Randal W.; Tartt, David; Antoniewicz, Robert F.; Agarwal, Arvind K.: The use of an automated flight test management system in the development of a rapid-prototyping flight research facility. In: *NASA Technical Memorandum 100435*, 1988, p. 1–21

[DJB1986] Duke, Eugene L.; Jones, Frank P.; Boncoli, Ralph B.: Development and Flight Test of an Experimental Maneuver Autopilot for a Highly Maneuverable Aircraft. In: *NASA Technical Paper 2618*, 1986, p. 1–61

[EAS2015a] EASA: A-NPA 2015-10 - Introduction of a regulatory framework for the operation of drones. (2015). https://www.easa.europa.eu/document-library/notices-of-proposed-amendment/npa-2015-10. [Accessed: 11.09.2019]

[EAS2015b] EASA: Concept of Operations for Drones. (2015). https://www.easa.europa.eu/document-library/general-publications/concept-operations-drones. [Accessed: 11.09.2019]

[EAS2015c] EASA: Opinion of a technical nature - Introduction of a regulatory framework for the operation of unmanned aircraft. (2015). https://www.easa.europa.eu/document-library/opinions/opinion-technical-nature. [Accessed: 11.09.2019]

[EAS2017a] EASA: NPA 2017-05 A - Introduction of a regulatory framework for the operation of drones — Unmanned aircraft system operations in the open and specific category. (2017). https://www.easa.europa.eu/document-library/notices-of-proposed-amendment/npa-2017-05. [Accessed: 11.09.2019]

[EAS2017b] EASA: NPA 2017-05 B - Introduction of a regulatory framework for the operation of drones — Unmanned aircraft system operations in the open and specific category. (2017). https://www.easa.europa.eu/document-library/notices-of-proposed-amendment/npa-2017-05. [Accessed: 11.09.2019]

[EAS2018] EASA: Opinion 01/2018 - Unmanned aircraft system (UAS) operations in the 'open' and 'specific' categories. (2018). https://www.easa.europa.eu/document-library/opinions/opinion-012018. [Accessed: 11.09.2019]

[EAS2019] EASA: Civil drones (Unmanned aircraft). (2019). https://www.easa.europa.eu/easa-and-you/civil-drones-rpas. [Accessed: 14.09.2019]

[Ele2018a] Elektra Solar: Elektra one Solar. (2018). https://www.elektra-solar.com/about. [Accessed: 02.07.2019]

[Ele2018b] Elektra Solar: Elektra one Solar. (2018). https://www.elektra-solar.com/products/elektra-one-solar. [Accessed: 02.07.2019]

[Erp2000] Erp, Jan B.: Controlling Unmanned Vehicles: the Human Factors Solution. In: *RTO SCI Symposium on "Warfare Automation Procedures and Techniques for Unmanned Vehicles*, 2000, p. 1–13

[Fra1993] France Ministry of Transport and Tourism: Official Report into the accident on 20 January 1992 near Mont Sainte-Odile of the Airbus A320 registered F-GGED operated by Air Inter. (1993). https://reports.aviation-safety.net/1992/19920120-0_A320_F-GGED.pdf. [Accessed: 21.02.2020]

[GHSM2021] Gabrys, Agnes; Holzapfel, Florian; Steffensen, Rasmus; Merkl, Christian: Flight Test Based Gain Tuning using non-parametric Frequency Domain Methods. In: *Scitech Forum 2021*, 2021, p. 1–15

[GH2012] Geiser, Markus; Heller, Matthias: Flight Dynamics Analysis and Basic Stabilization Study in Early Design Stages of the SAGITTA Demonstrator UAV. In: *DGLR, Deutscher Luft- und Raumfahrtkongress*, 2012, p. 1–8

[GSL+2018] Gierszewski, Daniel; Schneider, Volker; Lauffs, Patrick J.; Peter, Lars; Holzapfel, Florian: Clothoid-Augmented Online Trajectory Generation for Radius to Fix Turns. In: *15th IFAC Symposium on Control in Transportation Systems*, 2018, p. 174–179

[Gin1966] Ginsburg, Seymour: *The Mathematical Theory of Context-Free Languages.* 1966

[GS2004] Glenn H. Curtiss Museum; Scheck, Lieutenant Colonel W.: Lawrence Sperry: Genius on Autopilot. (2004). `https://www.historynet.com/lawrence-sperry-autopilot-inventor-and-aviation-innovator.htm`. [Accessed: 08.09.2019]

[HVCR2001] Hayhurst, Kelly J.; Veerhusen, Dan S.; Chilenski, John J.; Rierson, Leanna K.: A Practical Tutorial on Modified Condition/Decision Coverage. In: *NASA Technical Memorandum 210876*, 2001, p. 1–82

[Hen1968] Hennie, Frederick C.: *Finite-State Models for Logical Machines.* 1968

[HTA1991] Hewett, Marle; Tartt, David; Agarwal, Arvind K.: Automated flight test management system. In: *NASA Contractor Report 186011*, 1991, p. 1–44

[HHH2016] Hochstrasser, Markus; Hornauer, Markus; Holzapfel, Florian: Formal Verification of Flight Control Applications along a Model-Based Development Process: A Case Study. In: *Workshop Software Safety*, 2016, p. 1–41

[HSN+2017] Hochstrasser, Markus; Schatz, Simon P.; Nürnberger, Katejan; Hornauer, Markus; Myschik, Stephan; Holzapfel, Florian: Aspects of a Consistent Modeling Environment for DO-331 Design Model Development of Flight Control Algorithms. In: *CEAS EuroGNC*, 2017, p. 69–86

[Hol1982] Holcombe, Mike: *Algebraic automata theory.* Cambridge University Press, 1982. – ISBN 0521231965

[HMU2007] Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D.: *Introduction to Automata Theory, Languages, and Computation.* Gradiance Corp., 2007. – ISBN 0321455363

[Huf1954] Huffman, David A.: The synthesis of sequential switching circuits. In: *Journal of the Franklin Institute*, 1954, p. 161–190

[Hun1933] Huntington, Edward V.: A New Set of Independent Postulates for the Algebra of Logic with Special Reference to Whitehead and Russell's Principia Mathematica. In: *Transactions of the American Mathematical Society*, 1933, p. 274–304

[IEE1984] IEEE: *IEEE Std 91-1984.* 1984

[IEE1991] IEEE: IEEE Standard Graphic Symbols for Logic Functions (Including and incorporating IEEE Std 91a-1991, Supplement to IEEE Standard Graphic Symbols for Logic Functions). In: *IEEE Std 91a-1991*, 1991, p. 1–160

[Ive2015] Ives, Tucker: Ohio "Repudiates" Connecticut Over First-in-Flight Claims. (2015). https://www.wnpr.org/post/ohio-repudiates-connecticut-over-first-flight-claims. [Accessed: 31.05.2020]

[Jac2013] Jackson, Paul: Executive Overview: Jane's All the World's Aircraft: Development & Production. (2013). http://www.gustav-weisskopf.de/mediapool/93/932814/data/Artikel_Paul_Jackson.pdf. [Accessed: 15.03.2013]

[JAR2019] JARUS: JARUS guidelines on Specific Operations Risk Assessment (SORA). (2019). http://jarus-rpas.org/sites/jarus-rpas.org/files/jar_doc_06_jarus_sora_v2.0.pdf. [Accessed: 08.02.2020]

[Jat2006] Jategaonkar, Ravindra V.: *Flight vehicle system identification: A time domain methodology.* American Institute of Aeronautics and Astronautics, 2006. – ISBN 1563478366

[KH2005] Karim, Samin; Heinze, Clinton: Experiences with the design and implementation of an agent-based autonomous UAV controller. In: *AAMAS 2005, Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, 2005, p. 19–26

[KGSH2016] Karlsson, Erik; Gabrys, Agnes C.; Schatz, Simon P.; Holzapfel, Florian: Dynamic Flight Path Control Coupling for Energy and Maneuvering Integrity. In: *14th International Conference on Control, Automation, Robotics & Vision (ICARCV)*, 2016, p. 1–6

[KHB+2017] Karlsson, Erik; Holzapfel, Florian; Baier, Thaddäus; Dörhöfer, Christoph; Gabrys, Agnes C.; Hochstrasser, Markus; Krause, Christoph; Lauffs, Patrick J.; Mumm, Nils C.; Nürnberger, Katejan; Peter, Lars; Schatz, Simon P.; Schneider, Volker; Spiegel, Philip; Steinert, Lukas; Zollitsch, Alexander W.: Active Control Objective Prioritization for High-Bandwidth Automatic Flight Path Control. In: *CEAS EuroGNC*, 2017, p. 141–161

[KSB+2016] Karlsson, Erik; Schatz, Simon P.; Baier, Thaddäus; Dörhöfer, Christoph; Gabrys, Agnes C.; Hochstrasser, Markus; Krause, Christoph; Lauffs, Patrick J.; Mumm, Nils C.; Nürnberger, Katejan; Peter, Lars; Schneider, Volker; Spiegel, Philip; Steinert, Lukas; Zollitsch, Alexander W.; Holzapfel, Florian: Automatic Flight Path Control of an Experimental DA42 General Aviation Aircraft. In: *14th International Conference on Control, Automation, Robotics & Vision (ICARCV)*, 2016, p. 1–6

[KSH+2017] Karlsson, Erik; Schatz, Simon P.; Holzapfel, Florian; Baier, Thaddäus; Dörhöfer, Christoph; Gabrys, Agnes C.; Hochstrasser, Markus; Krause, Christoph; Lauffs, Patrick J.; Mumm, Nils C.; Nürnberger, Katejan; Peter, Lars; Schneider, Volker; Spiegel, Philip; Steinert, Lukas; Zollitsch, Alexander W.: Development of an Automatic Flight Path Controller for a DA42 General Aviation Aircraft. In: *CEAS EuroGNC*, 2017, p. 121–139

[Kel2001] Keller, Robert M.: *Computer Science: Abstraction to Implementation.* 2001

[KM2006] Klein, Vladislav; Morelli, Eugene A.: *Aircraft system identification: Theory and practice.* American Institute of Aeronautics and Astronautics, 2006. – ISBN 9781563478321

[KGH2018] Krause, Christoph; Göttlicher, Christoph; Holzapfel, Florian: Development of a generic Flight Test Maneuver Injection Module. In: *ICAS 31st Congress of the International Council of the Aeronautical Sciences (ICAS2018)*, 2018, p. 1–10

[KH2016] Krause, Christoph; Holzapfel, Florian: Designing a System Automation for a novel UAV Demonstrator. In: *14th International Conference on Control, Automation, Robotics & Vision (ICARCV)*, 2016, p. 1–6

[KH2017a] Krause, Christoph; Holzapfel, Florian: Designing and Implementing a Clutch Automation for a fully electric Fixed-Wing OPV. In: *2nd International Conference in Aerospace for Young Scientists (2nd ICAYS)*, 2017, p. 1–8

[KH2017b] Krause, Christoph; Holzapfel, Florian: Development of a generic Loiter Automation for a fixed wing UAV/OPV. In: *The 2017 Asian Control Conference - ASCC 2017*, 2017, p. 365–370

[KH2018a] Krause, Christoph; Holzapfel, Florian: Implementing a multi-level finite state machine with MATLAB Simulink and Stateflow in the environment of high-integrity aircraft controller software. In: *2018 4th International Conference on Control, Automation and Robotics (ICCAR)*, 2018, p. 147–151

[KH2018b] Krause, Christoph; Holzapfel, Florian: System Automation of a DA42 General Aviation Aircraft. In: *2018 Aviation Technology, Integration, and Operations Conference, AIAA AVIATION Forum*, 2018, p. 1–9

[KBS2011] Kriegel, Michael; Brueggenwirth, Stefan; Schulte, Axel: Knowledge Configured Vehicle - A layered artificial cognition based approach to decoupling high-level UAV mission tasking from vehicle implementation properties. In: *AIAA Guidance, Navigation, and Control Conference*, 2011, p. 1–19

[KHT2013] Krings, Matthias; Henning, Karsten; Thielecke, Frank: Flight Test Oriented Autopilot Design for Improved Aerodynamic Parameter Identification. In: *Advances in Aerospace Guidance, Navigation and Control*, 2013, p. 265–276

[Kuc2018] Kuchar, Richard O.: A versatile Simulation environment for Design Verification, System Integration Testing and Pilot Training of a diamond-shaped Unmanned Aerial Vehicle. In: *2018 AIAA Modeling and Simulation Technologies Conference*, 2018, p. 1–23

[KHH2018] Kügler, Martin E.; Heller, Matthias; Holzapfel, Florian: Automatic Take-off and Landing on the Maiden Flight of a Novel Fixed-Wing UAV. In: *2018 AIAA Flight Testing Conference*, 2018, p. 1–12

[KH2015] Kügler, Martin E.; Holzapfel, Florian: Enhancing the Auto Flight System of the SAGITTA Demonstrator UAV by Fault Detection and Diagnosis. In: *Aerospace Electronics and Remote Sensing Technology (ICARES), 2015 IEEE International*, 2015, p. 1–7

[KH2016a] Kügler, Martin E.; Holzapfel, Florian: Designing a Safe and Robust Automatic Take-off Maneuver for a Fixed-Wing UAV. In: *14th International Conference on Control, Automation, Robotics & Vision (ICARCV)*, 2016, p. 1–6

[KH2016b] Kügler, Martin E.; Holzapfel, Florian: Developing Automated Contingency Procedures for the ATOL System of a Fixed-Wing UAV through Online FDD. In: *AIAA Modeling and Simulation Technologies Conference*, 2016, p. 1–10

[KH2017a] Kügler, Martin E.; Holzapfel, Florian: Autoland for a Novel UAV as a State-Machine-based Extension to a Modular Automatic Flight Guidance and Control System. In: *The 2017 American Control Conference (ACC)*, 2017, p. 2231–2236

[KH2017b] Kügler, Martin E.; Holzapfel, Florian: Online Self-Monitoring of Automatic Take-off and Landing Control of a Fixed-Wing UAV. In: *IEEE Conference on Control Technology and Applications (CCTA)*, 2017, p. 2108–2113

[KH2017c] Kügler, Martin E.; Holzapfel, Florian: Parameterization and Computation of Automatic Take-off and Landing Trajectories for Fixed-Wing UAV. In: *17th AIAA Aviation Technology, Integration, and Operations Conference*, 2017, p. 1–11

[KH2018] Kügler, Martin E.; Holzapfel, Florian: Planning, Implementation, and Execution of an Automatic First Flight of a UAV. In: *ICAS 31st Congress of the International Council of the Aeronautical Science*, 2018, p. 1–8

[KSHH2018] Kügler, Martin E.; Seiferth, David; Heller, Matthias; Holzapfel, Florian: Real-Time Monitoring of Flight Tests with a Novel Fixed-Wing UAV by Automatic Flight Guidance and Control System Engineers. In: *2018 4th International Conference on Control, Automation and Robotics (ICCAR)*, 2018, p. 141–146

[LS2017] Lee, Edward A.; Seshia, Sanjit A.: *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. MIT Press, 2017. – ISBN 9780262533812

[Lio1996] Lions, Prof. Jacques-Louis: ARIANE 5 - Flight 501 Failure. (1996). `http://sunnyday.mit.edu/nasa-class/Ariane5-report.html`. [Accessed: 29.03.2020]

[LKN2011] Lisagor, Oleg; Kelly, Tim; Niu, Ru: Model-based safety assessment: Review of the discipline and its challenges. In: *The Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety*, 2011, p. 625–632

[Lit2019] Little, Becky: Automation of Planes Began 9 Years After the Wright Bros Took Flight — But It Still Leads to Baffling Disasters. (2019). https://www.history.com/news/plane-automation-autopilot-flight-302-610. [Accessed: 08.09.2019]

[Lud2019a] Ludwig Bölkow Campus: AURAIS. (2019). https://lb-campus.com/research/aurais. [Accessed: 03.07.2019]

[Lud2019b] Ludwig Bölkow Campus: EUROPAS. (2019). https://lb-campus.com/research/europas. [Accessed: 03.07.2019]

[Mar2019] MarketsAndMarkets: Unmanned Aerial Vehicle (UAV) Market - Global Forecast to 2025. (2019). https://www.marketsandmarkets.com/Market-Reports/unmanned-aerial-vehicles-uav-market-662.html. [Accessed: 12.02.2020]

[Mas2014] Master Films: A380 cockpit virtual visit. (2014). https://ccntservice.airbus.com/apps/cockpits/a380/. [Accessed: 25.09.2019]

[Mat2012] MathWorks Automotive Advisory Board: *Control Algorithm Modeling Guidelines using MATLAB Simulink, and Stateflow*. 2012

[Mat2016] MathWorks Automotive Advisory Board: Simulink - Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow. (2016). https://de.mathworks.com/help/releases/R2016b/pdf_doc/simulink/maab_guidelines.pdf. [Accessed: 10.04.2019]

[MW2004] McCarley, Jason S.; Wickens, Christopher D.: *Human Factors Concerns in UAV Flight*. 2004

[Mea1955] Mealy, George H.: A method for synthesizing sequential circuits. In: *The Bell System Technical Journal*, 1955, p. 1045–1079

[MPH2017] Meinecke, Verena; Prof. Hornung, Mirko: Unmanned flying wing successfully tested - Innovative technologies on board the unmanned aircraft Sagitta. (2017). https://www.tum.de/nc/en/studinews/issue-022018/show/article/detail/News/34147/. [Accessed: 03.07.2019]

[MWD1989] Menon, P. K. A.; Walker, R. A.; Duke, E. L.: Flight-test maneuver modeling and control. In: *Journal of Guidance, Control, and Dynamics*, 1989, p. 195–200

[MAW+2005] Miller, Steven; Anderson, Elise; Wagner, Lucas; Whalen, Michael; Heimdahl, Matts: Formal Verification of Flight Critical Software. In: *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2005, p. 1–16

[MH2003] Monin, Jean-François; Hinchey, Michael: *Understanding Formal Methods*. Springer, 2003. – ISBN 9781852332471

[Mon1948] Montgomerie, G. A.: Sketch for an algebra of relay and contactor circuits. In: *Journal of the Institution of Electrical Engineers - Part III: Radio and Communication Engineering*, 1948, p. 303–312

[Moo1956] Moore, Edward F.: Gedanken-Experiments on Sequential Machines. In: *Automata Studies*, 1956, p. 129–153

[Moo1965] Moore, Gordon E.: Cramming more components onto integrated circuits. In: *Electronics of the IEEE*, 1965, p. 1–4

[Mor1990] Morelli, Eugene A.: *Practical Input Optimization for Aircraft Parameter Estimation Experiments*, The George Washington University, Diss., 1990

[Mor2012a] Morelli, Eugene A.: Flight Test Maneuvers for Efficient Aerodynamic Modeling. In: *Journal of Aircraft*, 2012, p. 1857–1867

[Mor2012b] Morelli, Eugene A.: Real-Time Aerodynamic Parameter Estimation Without Air Flow Angle Measurements. In: *Journal of Aircraft*, 2012, p. 1064–1074

[MK1990] Morelli, Eugene A.; Klein, Vladislav: Optimal Input Design for Aircraft Parameter Estimation Using Dynamic Programming Principles. In: *17th Atmospheric Flight Mechanics Conference*, 1990, p. 1–11

[MH2017] Mumm, Nils C.; Holzapfel, Florian: Development of an Automatic Landing System for Diamond DA 42 aircraft utilizing a Load Factor Inner Loop Command System. In: *CEAS EuroGNC*, 2017, p. 86–91

[MKHS2017] Mumm, Nils C.; Kügler, Martin E.; Holzapfel, Florian; Schwithal, Alexander: C2LAND – Increasing Safety of Automatic Landing Systems for General Aviation Aircraft by Optical Runway Detection. In: *International Symposium on Precision Approach and Performance Based Navigation (ISPA)*, 2017, p. 1–6

[MSH2017] Mumm, Nils C.; Schatz, Simon P.; Holzapfel, Florian: Evaluation of the Flight Technical Error of a Trajectory Controller During Final Approach. In: *International Symposium on Precision Approach and Performance Based Navigation (ISPA)*, 2017, p. 1–6

[MSH2015] Mumm, Nils C.; Schneider, Volker; Holzapfel, Florian: Nonlinear continuous and differentiable 3D trajectory command generation. In: *Aerospace Electronics and Remote Sensing Technology (ICARES), 2015 IEEE International*, 2015, p. 1–9

[MZS+2017] Mumm, Nils C.; Zollitsch, Alexander W.; Schatz, Simon P.; Wulf, Simona; Holzapfel, Florian; Lauffs, Patrick J.; Peter, Lars: Design and Testing of a Ground Roll Runway Centerline Tracking Controller for a General Aviation Research Aircraft. In: *The 2017 Asian Control Conference - ASCC 2017*, 2017, p. 1689–1694

[Nor2001] Northrop Grumman: Globalhawk RQ-4 - Unmanned Aircraft System. (2001). https://www.northropgrumman.com/what-we-do/air/globalhawk-rq-4-unmanned-aircraft-system/. [Accessed: 10.02.2020]

[NHH2017] Nürnberger, Katejan; Hochstrasser, Markus; Holzapfel, Florian: Execution time analysis and optimisation techniques in the model-based development of a flight control software. In: *IET Cyber-Physical Systems: Theory & Applications*, 2017, p. 1–8

[Pet2017] Peter, Lars: DA42 MNG FBW Research Aircraft (Since 2008). In: *In-Flight Simulators and Fly-by-Wire/Light Demonstrators: A Historical Account of International Aeronautical Research*, 2017, p. 146–148

[PSJF2016] Pinchetti, Federico; Stephan, Johannes; Joos, Alexander; Fichter, Walter: FlySmart - Automatic Take-Off and Landing of an EASA CS-23 Aircraft. In: *Deutscher Luft- und Raumfahrtkongress 2016*, 2016, p. 1–9

[Pnu1977] Pnueli, Amir: The temporal logic of programs. In: *18th Annual Symposium on Foundations of Computer Science*, 1977, p. 46–57

[Rei2017] Reichel, Reinhard: DIAMOND DA42—FlySmart-FBW23 (2012–2015). In: *In-Flight Simulators and Fly-by-Wire/Light Demonstrators: A Historical Account of International Aeronautical Research*, 2017, p. 148–150

[RK2004] Roth, Charles H.; Kinney, Larry L.: *Fundamentals of Logic Design*. Cengage Learning, 2004. – ISBN 0495668044

[Roz2011] Rozier, Kristin: Linear Temporal Logic Symbolic Model Checking. In: *Computer Science Review*, 2011, p. 163–203

[RTC1992] RTCA: *RTCA DO-178 - Software Aspects and Considerations in Airborne Related Equipment.* 1992

[RTC2011a] RTCA: *RTCA DO-178 - Software Considerations in Airborne Systems and Equipment Certification.* 2011

[RTC2011b] RTCA: *RTCA DO-331 - Model-Based Development and Verification Supplement to DO-178C and DO-278A.* 2011

[RTC2014] RTCA: *RTCA-DO 236C Minimum Aviation System Performance Standards: Required Navigation Performance for Area Navigation.* 2014

[SW1994] Sarter, Nadine; Woods, David: Pilot Interaction With Cockpit Automation II: An Experimental Study of Pilots' Model and Awareness of the Flight Management System. In: *International Journal of Aviation Psychology*, 1994, p. 1–28

[SGGH2018] Schatz, S. P.; Gabrys, A. C.; Gierszewski, D. M.; Holzapfel, F.: Inner Loop Command Interface in a Modular Flight Control Architecture for Trajectory Flights of General Aviation Aircraft. In: *2018 5th International Conference on Control, Decision and Information Technologies (CoDIT)*, 2018, p. 86–91

[SH2014] Schatz, Simon P.; Holzapfel, Florian: Modular trajectory / path following controller using nonlinear error dynamics. In: *Aerospace Electronics and Remote Sensing Technology (ICARES), 2014 IEEE International*, 2014, p. 157–163

[SH2017] Schatz, Simon P.; Holzapfel, Florian: Nonlinear Modular 3D Trajectory Control of a General Aviation Aircraft. In: *CEAS EuroGNC*, 2017, p. 163–183

[SSK+2016] Schatz, Simon P.; Schneider, Volker; Karlsson, Erik; Holzapfel, Florian; Baier, Thaddäus; Dörhöfer, Christoph; Hochstrasser, Markus; Gabrys, Agnes C.; Krause, Christoph; Lauffs, Patrick J.; Mumm, Nils C.; Nürnberger, Katejan; Peter, Lars; Spiegel, Philip; Steinert, Lukas; Zollitsch, Alexander W.: Flightplan Flight Tests of an Experimental DA42 General Aviation Aircraft. In: *14th International Conference on Control, Automation, Robotics & Vision (ICARCV)*, 2016, p. 1–6

[SH2017] Schneider, Volker; Holzapfel, Florian: Modular Trajectory Generation Test Platform for Real Flight Systems. In: *CEAS EuroGNC*, 2017, p. 185–202

[SMH2015] Schneider, Volker; Mumm, Nils C.; Holzapfel, Florian: Trajectory generation for an integrated mission management system. In: *Aerospace Electronics and Remote Sensing Technology (ICARES), 2015 IEEE International*, 2015, p. 1–7

[SPS+2016] Schneider, Volker; Piprek, Patrick; Schatz, Simon P.; Baier, Thaddäus; Dörhöfer, Christoph; Hochstrasser, Markus; Gabrys, Agnes C.; Karlsson, Erik; Krause, Christoph; Lauffs, Patrick J.; Mumm, Nils C.; Nürnberger, Katejan; Peter, Lars; Spiegel, Philip; Steinert, Lukas; Zollitsch, Alexander W.; Holzapfel, Florian: Online Trajectory Generation Using Clothoid Segments. In: *14th International Conference on Control, Automation, Robotics & Vision (ICARCV)*, 2016, p. 1–6

[SKH2017] Seiferth, D.; Kuchar, R.; Heller, M.: Model-based design and real live on-runway testing of a ground controller for a novel diamond-shaped Unmanned Air Vehicle (UAV). In: *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, 2017, p. 3934–3941

[SH2017] Seiferth, David; Heller, Matthias: Testing and performance enhancement of a model-based designed ground controller for a diamond-shaped unmanned air vehicle (UAV). In: *IEEE Conference on Control Technology and Applications (CCTA)*, 2017, p. 1988–1994

[SKHH2018] Seiferth, David; Kügler, Martin E.; Heller, Matthias; Holzapfel, Florian: In-Flight Verification of a model-based designed Ground Controller for an innovative Unmanned Air Vehicle (UAV). In: *2018 AIAA Flight Testing Conference*, 2018, p. 1–12

[Sha1938] Shannon, Claude E.: A symbolic analysis of relay and switching circuits. In: *Electrical Engineering*, 1938, p. 713–723

[SM1956] Shannon, Claude E.; McCarthy, John: *Automata Studies - Preface.* 1956

[She1992] Sheridan, Thomas B.: *Telerobotics, Automation, and Human Supervisory Control.* M.I.T. Press, 1992. – ISBN 9780262193160

[Sig2016] Sigler, Dean: Acentiss Electric Dual-Rotor Aviation Motor. (2016). `http://sustainableskies.org/acentiss-electric-dual-rotor-aviation-motor/`. [Accessed: 02.07.2019]

[Sim1999] Simon, Matthew: *Automata Theory*. World Scientific Publishing Co. Pte. Ltd., 1999. – ISBN 9810237537

[Sip2006] Sipser, Michael: *Introduction to the Theory of Computation*. Thomson Course Technology, 2006. – ISBN 0534950973

[Smi2003] Smithsonian: The Wright Brothers - The Invention of the Aerial Age. (2003). https://airandspace.si.edu/exhibitions/wright-brothers/online/. [Accessed: 08.09.2019]

[Soc2018] Society of Automotive Engineers: Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles. (2018). https://www.sae.org/standards/content/j3016_201806/. [Accessed: 19.10.2019]

[Sto2010] Storm, Walter: Solving Sudoku using Simulink Design Verifier-A Model Checking Example. In: *AIAA Infotech at Aerospace 2010*, 2010, p. 1–4

[Str2018] Strauch, Barry: Ironies of Automation: Still Unresolved After All These Years. In: *IEEE Transactions on Human-Machine Systems*, 2018, p. 419–433

[TM2014] The MathWorks, Inc.: Unit Testing Framework. (2014). https://de.mathworks.com/help/releases/R2014a/matlab/matlab-unit-test-framework.html. [Accessed: 09.03.2020]

[TM2016a] The MathWorks, Inc.: All Products. (2016). https://de.mathworks.com/help/releases/R2016b/. [Accessed: 09.03.2020]

[TM2016b] The MathWorks, Inc.: Call C Functions in C Charts. (2016). https://de.mathworks.com/help/releases/R2016b/stateflow/ug/calling-c-functions-in-actions.html. [Accessed: 16.03.2020]

[TM2016c] The MathWorks, Inc.: Chart Programming Basics - Guidelines for building Stateflow charts. (2016). https://de.mathworks.com/help/releases/R2016b/stateflow/state-chart-programming-basics.html. [Accessed: 10.04.2019]

[TM2016d] The MathWorks, Inc.: Control Chart Execution Using Temporal Logic. (2016). https://www.mathworks.com/help/releases/R2016b/stateflow/ug/using-temporal-logic-in-state-actions-and-transitions.html. [Accessed: 10.04.2019]

[TM2016e] The MathWorks, Inc.: Differences Between MATLAB and C as Action Language Syntax. (2016). https://de.mathworks.com/help/releases/R2016b/stateflow/ug/differences-between-matlab-and-stateflow-action-language.html. [Accessed: 16.03.2020]

[TM2016f] The MathWorks, Inc.: DO-178 Qualification Kit: - Model-Based Design Workflow for DO-178C. (2016). https://de.mathworks.com/products/do-178/features.html. [Accessed: 10.04.2019]

[TM2016g] The MathWorks, Inc.: Embedded Coder - Getting Started Guide. (2016). https://de.mathworks.com/help/releases/R2016b/pdf_doc/ecoder/ecoder_gs.pdf. [Accessed: 17.08.2019]

[TM2016h] The MathWorks, Inc.: Embedded Coder - User's Guide. (2016). https://de.mathworks.com/help/releases/R2016b/pdf_doc/ecoder/ecoder_ug.pdf. [Accessed: 17.08.2019]

[TM2016i] The MathWorks, Inc.: MATLAB - Desktop Tools and Development Environment. (2016). https://de.mathworks.com/help/releases/R2016b/pdf_doc/matlab/matlab_env.pdf. [Accessed: 09.03.2020]

[TM2016j] The MathWorks, Inc.: MATLAB - Primer. (2016). https://de.mathworks.com/help/releases/R2016b/pdf_doc/matlab/getstart.pdf. [Accessed: 09.03.2020]

[TM2016k] The MathWorks, Inc.: MATLAB - Release Notes. (2016). https://de.mathworks.com/help/releases/R2016a/pdf_doc/matlab/rn.pdf. [Accessed: 09.03.2020]

[TM2016l] The MathWorks, Inc.: MATLAB - Release Notes. (2016). https://de.mathworks.com/help/releases/R2016b/pdf_doc/matlab/rn.pdf. [Accessed: 09.03.2020]

[TM2016m] The MathWorks, Inc.: Model Architecture Guidelines. (2016). https://de.mathworks.com/help/releases/R2016b/simulink/model-architecture.html. [Accessed: 10.04.2019]

[TM2016n] The MathWorks, Inc.: Model Architecture Guidelines Stateflow. (2016). https://www.mathworks.com/help/releases/R2016b/simulink/simulink.html. [Accessed: 09.03.2020]

[TM2016o] The MathWorks, Inc.: Model Architecture Guidelines Stateflow. (2016). https://de.mathworks.com/help/releases/R2016b/simulink/stateflow.html. [Accessed: 09.03.2020]

[TM2016p] The MathWorks, Inc.: Polyspace Bug Finder. (2016). https://www.mathworks.com/help/releases/R2016b/bugfinder/index.html. [Accessed: 09.03.2020]

[TM2016q] The MathWorks, Inc.: Polyspace Code Prover. (2016). https://www.mathworks.com/help/releases/R2016b/codeprover/index.html. [Accessed: 09.03.2020]

[TM2016r] The MathWorks, Inc.: Simulink - Code Inspector - User's Guide. (2016). https://de.mathworks.com/help/releases/R2016b/pdf_doc/slci/slci_ug.pdf. [Accessed: 10.04.2019]

[TM2016s] The MathWorks, Inc.: Simulink - Design Verifier - User's Guide. (2016). https://de.mathworks.com/help/releases/R2016b/pdf_doc/sldv/sldv_ug.pdf. [Accessed: 17.08.2019]

[TM2016t] The MathWorks, Inc.: Simulink - Modeling Guidelines for Code Generation. (2016). https://de.mathworks.com/help/releases/R2016b/pdf_doc/simulink/cg_guidelines.pdf. [Accessed: 10.04.2019]

[TM2016u] The MathWorks, Inc.: Simulink - Modeling Guidelines for High-Integrity Systems. (2016). https://de.mathworks.com/help/releases/R2016b/pdf_doc/simulink/hi_guidelines.pdf. [Accessed: 10.04.2019]

[TM2016v] The MathWorks, Inc.: Simulink - User's Guide. (2016). https://de.mathworks.com/help/releases/R2016b/pdf_doc/simulink/sl_using.pdf. [Accessed: 17.08.2019]

[TM2016w] The MathWorks, Inc.: Simulink Coder - Getting Started Guide. (2016). https://de.mathworks.com/help/releases/R2016b/pdf_doc/rtw/rtw_gs.pdf. [Accessed: 09.03.2020]

[TM2016x] The MathWorks, Inc.: Stateflow - User's Guide. (2016). https://de.mathworks.com/help/releases/R2016b/pdf_doc/stateflow/sf_ug.pdf. [Accessed: 17.08.2019]

[TM2019] The MathWorks, Inc.: Types of Model Coverage. (2019). https://de.mathworks.com/help/slcoverage/ug/types-of-model-coverage.html. [Accessed: 07.03.2020]

[TM2020a] The MathWorks, Inc.: MathWorks - Accelerating the pace of engineering and science. (2020). https://www.mathworks.com/. [Accessed: 21.03.2020]

[TM2020b] The MathWorks, Inc.: MATLAB. (2020). https://www.mathworks.com/help/releases/R2020b/matlab/index.html. [Accessed: 22.09.2020]

[TM2020c] The MathWorks, Inc.: MATLAB - Getting Started Guide. (2020). https://www.mathworks.com/help/releases/R2020b/matlab/getting-started-with-matlab.html. [Accessed: 22.09.2020]

[TM2020d] The MathWorks, Inc.: Simulink - Getting Started Guide. (2020). https://www.mathworks.com/help/releases/R2020b/pdf_doc/simulink/simulink_gs.pdf. [Accessed: 22.09.2020]

[TM2020e] The MathWorks, Inc.: Stateflow - Getting Started Guide. (2020). https://www.mathworks.com/help/releases/R2020b/pdf_doc/stateflow/stateflow_gs.pdf. [Accessed: 22.09.2020]

[TR2012] Tischler, Mark B.; Remple, Robert K.: *Aircraft and rotorcraft system identification: Engineering methods with flight test examples*. American Institute of Aeronautics and Astronautics, 2012. – ISBN 1600868207

[Tur1936] Turing, Alan M.: On Computable Numbers, with an Application to the Entscheidungsproblem. In: *Proceedings of the London Mathematical Society*, 1936, p. 230–265

[Wie1989] Wiener, Earl L.: Human factors of advanced technology (glass cockpit) transport aircraft. In: *NASA Contractor Report 177528*, 1989, p. 1–220

[WC1980] Wiener, Earl L.; Curry, Renwick E.: Flight-Deck Automation: Promises and Problems. In: *NASA Technical Memorandum 81206*, 1980, p. 1–24

[Wie1948] Wiener, Norbert: *Cybernetics or Control and Communication in the Animal and the Machine*. 1948

[Wri1903a] Wright: Photos of the Wright brothers first airplane flight. (1903). http://www.wright-house.com/wright-brothers/wrights/wright-flyer.html. [Accessed: 11.09.2019]

[Wri1903b] Wright, Orville: Success four flights. (1903). https://www.wdl.org/en/item/11372/. [Accessed: 08.09.2019]

[Zim2017] Zimmermann: Erfolgreicher Erstflug von Sagitta. (2017). https://www.flugrevue.de/militaer/airbus-uav-demonstrator-erfolgreicher-erstflug-von-sagitta/. [Accessed: 03.07.2019]

[ZMW+2017] Zollitsch, Alexander W.; Mumm, Nils C.; Wulf, Simona; Holzapfel, Florian; Hochstrasser, Markus; Lauffs, Patrick J.; Peter, Lars: Automatic Takeoff of a General Aviation Research Aircraft. In: *The 2017 Asian Control Conference - ASCC 2017*, 2017, p. 1683–1688

[ZSMH2018] Zollitsch, Alexander W.; Schatz, Simon P.; Mumm, Nils C.; Holzapfel, Florian: Model-in-the-Loop Simulation of Experimental Flight Control Software. In: *2018 AIAA Modeling and Simulation Technologies Conference*, 2018, p. 1–18