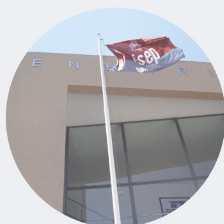
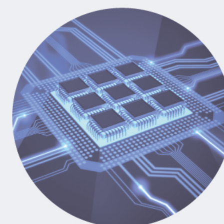
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA DE SISTEMAS COMPUTACIONAIS CRÍTICOS

isep





Análise Formal de um Protocolo de Transferência de Autoridade para VANTs

RICARDO ALEXANDRE ALMEIDA RODRIGUES
outubro de 2024

P.PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Formal Analysis of an Authority Transfer Protocol for UAVs

Ricardo Alexandre Almeida Rodrigues

Dissertation submitted in partial fulfilment of the requirements for the Master's degree in Critical Computing Systems Engineering

Supervisor: David Miguel Ramalho Pereira

Evaluation Committee:
President:
Luís Miguel Pinho, Professor Coordenador, Instituto Superior de Engenharia do Porto

Members:
David Miguel Ramalho Pereira, Investigador, Instituto Superior de Engenharia do Porto
Jorge Manuel Neves Coelho, Professor Adjunto, Instituto Superior de Engenharia do Porto

Porto, October 15, 2024

Statement of Integrity

I hereby declare having conducted this academic work with integrity. I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

I declare that the work presented in this document is original and my own, and has not previously been used for any other purpose.

I further declare that I have fully followed the Code of Good Practices and Conduct of the Polytechnic Institute of Porto.

ISEP, Porto, October 15, 2024

Dedicatory

To David, my supervisor, for your insight, dedication, and for seeing this journey through to the very end.

To my parents, for their unwavering love, support, and guidance throughout my life.

To Nuno, my colleague and friend, for being with me every step of the way as we navigated this academic journey together. Your companionship made this experience all the more meaningful.

To Mia, whose strength and love have been a constant source of inspiration. You've stood by my side since the start of my academic journey and lifted me when things seemed impossible. Your unwavering belief in me and your endless support mean more than words can express. I couldn't have made it this far without your encouragement and compassion. I will always treasure you as long as I live.

Abstract

The use of Unmanned Aerial Vehicles (UAVs) in a variety of applications has resulted in a growing demand for Beyond Visual Line of Sight (BVLOS) operations to cover long distances.

This thesis focuses on an initial analysis of a UAV communication protocol designed for transferring control authority between ground controls within a common range, addressing challenges and evaluating its efficacy across different operational phases. The research encompasses start-up, mission execution, and handover phases, assessing the protocol's interactions with ground-based systems for reliability, efficiency, and adaptability.

A comprehensive overview of model checking is provided, emphasizing transition systems, formalisms, and tools for system correctness verification. The research employs model checking to formally specify and model the UAV communication protocol, establishing evaluation criteria through the use of LTL and CTL properties. The derived requirements serve as a foundational framework for future iterations, ensuring the protocol's robustness and addressing security strategies in the event of connection loss. The holistic approach contributes to a comprehensive understanding of the protocol's functionality, aiming to enhance the security authorization handover procedure and promote public and regulatory acceptance of BVLOS operations with UAVs.

Keywords: Model Checking, UAV, Verification, Communication, Protocol

Resumo

A utilização de Veículos Aéreos Não Tripulados (VANTs) em diversas aplicações tem gerado uma procura crescente por operações Além da Linha de Visão Visual (BVLOS) para cobrir longas distâncias.

Esta investigação foca-se numa análise inicial de um protocolo de comunicação de VANT concebido para transferir a autoridade de controlo entre estações de controlo em terra dentro de um alcance comum, abordando desafios e avaliando a sua eficácia ao longo das fases operacionais. A investigação abrange as fases de arranque, execução da missão e transferência, avaliando as interações do protocolo com sistemas em terra em termos de fiabilidade, eficiência e adaptabilidade.

É apresentada uma visão abrangente da verificação de modelos, focando em sistemas de transição, formalismos e ferramentas para a verificação da correção do sistema. A investigação utiliza a verificação de modelos para especificar e modelar formalmente o protocolo de comunicação de VANT, estabelecendo critérios de avaliação através de propriedades LTL e CTL. Os requisitos derivados, específicos para o protocolo e abrangendo as fases operacionais, execução da missão e transferência, servem como uma base fundamental para futuras iterações, garantindo a robustez do protocolo e abordando estratégias de segurança em caso de perda de ligação. A abordagem holística contribui para uma compreensão extensiva da funcionalidade do protocolo, visando melhorar o procedimento de transferência de autorização de segurança e promover a aceitação pública e regulatória de operações BVLOS com VANTs.

Contents

List of Figures **xv**

List of Abbreviations **xvii**

1 Introduction **1**
1.1 Context 1
1.2 Document Structure 2

2 State of the Art **3**
2.1 Model Checking 3
2.1.1 Transition Systems 3
2.1.2 Linear Temporal Logic (LTL) 4
2.1.3 Computation Tree Logic (CTL) 5
2.1.4 Traffic Light implementation in NuSMV and LTL/CTL property assertion 6
LTL Properties 7
CTL Properties 9
Crucial differences 10
2.2 Probabilistic model checking 11
2.3 Model checking in UAVs 12
2.3.1 UAV monitoring road conditions in dangerous environment 12
2.3.2 Probabilistic Model Checking and Autonomy 12
2.4 Model checking in protocols 13
2.4.1 Secure Group Communication Protocol Verification 13
2.4.2 Model Checking in Large Network Protocol Implementations 14
2.4.3 Comprehensive Approach to Critical Avionics Systems Verification . 14
2.5 Tools 14
2.5.1 SPIN (Simple Promela Interpreter) 15
2.5.2 NuSMV (New Symbolic Model Verifier) 16
2.5.3 PRISM 18
2.5.4 UPPAAL 19

3 Analysis **21**
3.1 Scenario Definition 21
3.2 On Start-up 22
3.3 On Mission 22
3.4 On Handover 23
3.5 Ground Control 24
3.6 Derived Requirements 29
3.6.1 UAV Requirements 29
Start-up 29

Operating - Telemetry Information . . . 29
Operating - Message Handling . . . 29
3.6.2 Ground Control . . . 30
Ground Control - Start-up . . . 30
Ground Control - Message Handling . . . 30
Ground Control - Command Handling . . . 31
3.7 Conclusions . . . 31

4 Methodology **33**
4.1 Research approach . . . 33
4.2 Evaluation . . . 33

5 Model Checking Approach **35**
5.1 First Iteration of the model . . . 35
5.1.1 Unmanned aerial vehicle (UAV) module . . . 35
Initialization . . . 36
State Transitions . . . 37
Properties . . . 37
5.1.2 Ground control station (GCS) module . . . 40
Initialization . . . 41
State Transitions . . . 42
Properties . . . 42
5.1.3 Module main . . . 46
Initialization . . . 46
State Transitions . . . 47
Properties . . . 47
5.1.4 Results . . . 47
5.2 Second Iteration of the model . . . 48
5.2.1 Derived Requirements . . . 48
Unmanned Aerial Vehicle . . . 48
Ground Control Station . . . 50
5.2.2 NuSMV model . . . 50
Module main . . . 50
Module uav . . . 52
Module gcs . . . 52
Model Properties . . . 53
5.2.3 Results . . . 56
5.3 Third Iteration of the model . . . 57
5.3.1 Derived Requirements . . . 57
Unmanned Aerial Vehicle . . . 57
Ground Control Station . . . 57
5.3.2 NuSMV model . . . 57
Module main . . . 57
Module uav . . . 59
Module gcs . . . 60
Model Properties . . . 61
5.3.3 Results . . . 64

6 Conclusions **65**

A Appendix A - Iteration 1 NuSMV Model **67**

B Appendix B - Iteration 2 NuSMV model **87**

C Appendix C - Iteration 3 NuSMV model **93**

Bibliography **99**

List of Figures

2.1 Example for a transition system (Pereira and Bragança 2022) 4
2.2 Simple example of a Promela model. 15
2.3 NuSMV visual representation of the code example 16
2.4 Example of NuSMV code (Pereira and Bragança 2022) 17
2.5 UPPAAL coin example available as an example template in the program . . 19

3.1 Authority Handover Procedure available in (Ferreira 2023) 21
3.2 Procedure available in (Ferreira 2023) 23
3.3 Handover Procedure available in (Ferreira 2023) 24
3.4 Ground Control Procedure available in (Ferreira 2023) 25
3.5 Ground Control Commands available in (Ferreira 2023) 26
3.6 Ground Control Messages available in (Ferreira 2023) 27
3.7 Temporal diagram of the handover procedure available in (Ferreira 2023) . 28

5.1 Module uavControl definition 35
5.2 NuSMV Initialization of Module uavControl in Module main 37
5.3 Module GcsControl definition 40
5.4 NuSMV Initialization of Module GcsControl in Module main 42
5.5 Handover procedure available in (Ferreira 2023) 48
5.6 Handover procedure action diagram available in (Ferreira 2023) 49

List of Abbreviations

ACTL	**A**ction-based **C**omputational **T**ree **L**ogic
BDDs	**B**inary **D**ecision **D**iagrams
BVLOS	**B**eyond **V**isual **L**ine **o**f **S**ight
CSL	**C**ontinuous **S**tochastic **L**ogic
CTL	**C**omputation **T**ree **L**ogic
CTL*	**C**omputational **T**ree **L**ogic with **F**ixpoints
CTL-FO	**C**omputational **T**ree **L**ogic with **F**irst-**O**rder Logic
DTMC	**D**iscrete-**T**ime **M**arkov **C**hains
EARS	**E**asy **A**pproach to **R**equirements **S**yntax
GCS	**G**round **C**ontrol **S**tation
LTL	**L**inear **T**emporal **L**ogic
MTBDDs	**M**ulti-**T**erminal **B**inary **D**ecision **D**iagrams
MTL	**M**etric **T**emporal **L**ogic
NuSMV	**N**ew **S**ymbolic **M**odel **V**erifier
PCTL	**P**robabilistic **C**omputation **T**ree **L**ogic
PCTL*	**P**robabilistic **C**omputation **T**ree **L**ogic **E**xtended
PTL	**P**ropositional **T**emporal **L**ogic
SPIN	**S**imple **P**romela **In**terpreter
TPTL	**T**imed **P**ropositional **T**emporal **L**ogic
UAV	**U**nmanned **A**erial **V**ehicle

Chapter 1

Introduction

The rapid integration of Unmanned Aerial Vehicles (UAVs) across various applications has given rise to an escalating demand for Beyond Visual Line of Sight (BVLOS) operations, especially for covering extensive distances. At the core of BVLOS operations lies a reliance on data derived from onboard instruments, where the flight controller assumes a pivotal role in the control of drone missions and the acquisition of sensor data. Critical aircraft information, including position, altitude, speed, and flight direction, is conveyed via a radio link to either an operator or a Ground Control Station (GCS).

This thesis undertakes a comprehensive exploration of an UAV communication protocol tailored to facilitate the transfer of control authority between ground controls within a shared operational range. The primary focus of this thesis is to execute an initial analysis, address inherent challenges, and evaluate the protocol's efficacy throughout its various operational phases, namely start-up, mission execution, and handover. The assessment extends to the protocol's interactions with ground-based systems, emphasizing aspects of reliability, efficiency, and adaptability.

Within this context, a detailed overview of model checking is presented, emphasizing the significance of transition systems, formalisms, and tools for ensuring system correctness. The research employs model checking techniques to formally specify and model the UAV communication protocol, establishing clear evaluation criteria and a planned research timeline. The derived requirements, tailored specifically to the protocol, including start-up, mission execution, and handover phases, provide a robust foundation for successive iterations, ensuring the protocol's resilience and addressing security strategies in the event of a connection loss.

This holistic approach aims to deepen our understanding of the protocol's functionality, ultimately contributing to the enhancement of security authorization handover procedures and fostering widespread acceptance of BVLOS operations with UAVs, both within the public and regulatory frameworks.

1.1 Context

The surge in UAV applications, spanning aerial imaging to delivery services, has driven the need for efficient communication protocols, especially in BVLOS scenarios. This research contextualizes the critical intersection of UAVs and model checking protocols, aiming to address the challenges inherent in the UAV authority transfer protocol. Emphasis is placed on understanding its behavior during operational phases and its interaction with ground control systems.

1.2 Document Structure

This document is organized as follows:

Chapter 1: This chapter introduces the focus of the thesis and provides context to model checking protocols and UAVs as well as describing the document structure.

Chapter 2: This chapter provides an in-depth state of the art delving into transition systems and diverse model checking models, along with supporting tools. An examination of the application of model checking in UAVs is performed, with a focus on scenarios such as UAV monitoring road conditions in dangerous environments and the integration of probabilistic model checking with autonomy. Additionally, the chapter explores the role of model checking in protocols, covering secure group communication protocol verification, model checking in large network protocol implementations, and a comprehensive approach to critical avionics systems verification. The chapterconcludes with an overview of tools utilized in model checking, including the Simple Promela Interpreter (SPIN), New Symbolic Model Verifier (NuSMV), PRISM, and UPPAAL. The detailed textual description aims to provide a comprehensive understanding of the state of the art in model checking.

Chapter 3: This chapter presents the focal point of the dissertation - the analysis of the UAV authority transfer protocol. The primary emphasis is placed on analysing the intricacies and functionalities of the specific communication protocol employed in UAVs. The analysis encompasses a thorough exploration of the protocol's behaviour during different operational phases, including start-up, mission execution, and handover. This examination involves evaluating the protocol's effectiveness in facilitating communication, addressing potential challenges, and ensuring the guarantee of the transitions between phases. Furthermore, the analysis extends to ground control operations, emphasizing how the communication protocol interfaces with and serves the needs of ground-based control systems. This includes considerations of reliability, efficiency, and adaptability in diverse operational scenarios. Derived requirements specific to the UAV communication protocol are defined which includes a definition of the requirements during the loading phase, active mission execution, and the handover phase.

Chapter 4: This chapter defines the approach to investigate the protocol through the use of model checking and how will the protocol be evaluated.

Chapter 5: This chapter is structured to provide a a comprehensive understanding of the model and related specifications, as well as the iterative process of refinement. The model is initially present based on predefined derived requirements, detailing its components and functionality, followed by the specifications in Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). Each iteration concludes with a summary of the insights and the required actions for the next iteration.

Chapter 6: Finally, this chapter synthesizes the key findings in this dissertation. The chapter revisits the initial objectives and evaluates how they were achieved.

Chapter 2

State of the Art

In this chapter is present a comprehensive study into the state of the art of model checking, addressing transition systems, different types of model checking formalism and supporting tools.

2.1 Model Checking

Model checking is a formal technique that verifies a system's correctness by exhaustively checking all possible states against a set of properties and specifications. It entails developing a mathematical model of the system's behavior, generating a finite-state representation, and checking if the model satisfies logical formulas representing desired properties (Baier and Katoen 2008).

The foundation of model checking is established in this section, with a focus on state machines as a key component. The process involves developing a mathematical model, creating a finite-state representation, and methodically determining whether the model complies with logical formulas that capture desired properties.

2.1.1 Transition Systems

A transition system is a mathematical abstraction that depicts a system's evolution through a series of states and transitions between them. The concept of transition systems (Baier and Katoen 2008), which serves as a fundamental construct for representing the dynamic behavior of systems, is central to model checking.

Normally, a transition system is composed by the following components/elements:

- **States:** In the context of model checking, states are distinct configurations that the system can occupy. These configurations encapsulate relevant information about the system at a specific point in time, including variables, parameters, and other pertinent attributes.
- **Transitions:** Transitions define the possible movements or changes between states. They represent the actions, events, or processes that drive the system from one state to another. Each transition is associated with a condition or set of conditions that determine when it can be triggered.
- **Initial and Final States:** A model typically designates an initial state from which the system starts its execution. Additionally, final states represent configurations that satisfy specified properties or conditions, serving as the desired outcomes of the verification process.

- **Formal Representation and Logical Formulas:** The process of model checking involves the creation of a formal, mathematical model of the system, often represented as a state transition system. This model encapsulates the system's behaviors and interactions in a structured manner, enabling a systematic analysis of its properties.

The desired system properties are specified using logical formulas expressed in temporal logic or other formal languages. These formulas express constraints, requirements, or invariants that the system must follow in order for verification to succeed.

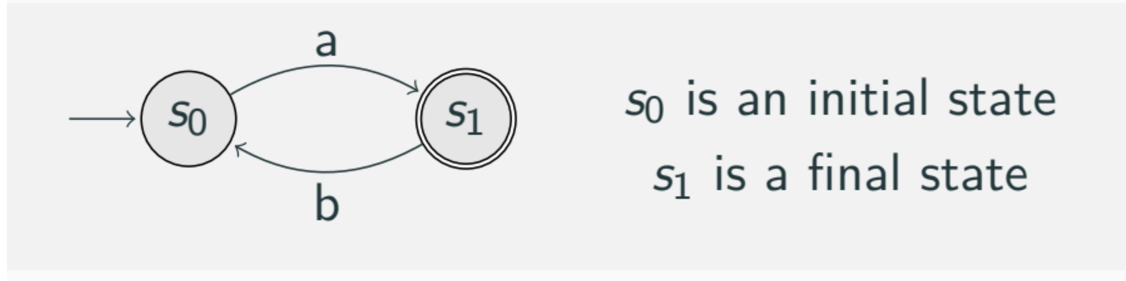


Figure 2.1: Example for a transition system (Pereira and Bragança 2022)

The states s_0 and s_1 are states of a system. For the example presented in figure 2.1, the initial state of the system is s_0 and the final state is s_1. The transition from s_0 to s_1 is denominated as a and the transition from s_1 to s_0 is denominated as b.

If the system is in state s_0 and b occurs, the system shall remain in the same state s_0. The same applies for both state s_1 and transition a.

If the system is in state s_0 and a occurs, the system shall transition to the state s_1. If the system is in state s_1 and b occurs, the system shall transition to the state s_0.

The formalization of the example in figure 2.1 can be represented as a tuple $< S, I, \downarrow, \rightarrow >$. To represent a transition system, the following notation can be used:

- $S = \{s_0, s_1\}$: Definition of the set of possible states.
- $I = \{s_0\}$: Definition of the initial state of the system.
- $\downarrow = \{s_1\}$: Definition of the final state of the system.
- $\rightarrow = \{< s_0, a, s_1 >, < s_0 1, b, s_0 >\}$: Definition of the transition relation between states.
- $\neg$ = Definition of negation of a state.
- V = Definition of *AND* in an expression of a property.
- | = Definition of *OR* in an expression of a property.

2.1.2 Linear Temporal Logic (LTL)

Linear Temporal Logic (Baier and Katoen 2008) is a formalism used in model checking to express and verify temporal properties of systems. LTL extends propositional logic with temporal operators, allowing the specification of properties over sequences of states, commonly known as execution traces. LTL is particularly useful in reasoning about the behavior of reactive systems, which are systems that continuously interact with their environment.

Propositional logic deals with statements that are either true or false, and it includes logical connectives such as AND, OR, and NOT.

- $\circ$ **(Circle):** Read as "next step". For example $\circ$ p means "In the next step, p will be true.
- $\Diamond$ **(Diamond):** Read as "eventually" or "sometime in the future". For example, $\Diamond$ p means "eventually, p is true".
- $\Box$ **(Box):** Read as "always". For example, $\Box$p means "always, p is true".
- U **(Until):** Binary operator denoted as p U q, meaning "p holds until q is true". It indicates that p is true until the condition q becomes true.
- $\models$ **(Satisfaction relation):** Binary relation between a model and a formula that indicates whether the formula is satisfied in the model.
- **Linear Structure:** LTL considers linear sequences of states or time points. The "linear" aspect implies that the temporal relationships are along a single, well-ordered timeline.

LTL formulas are constructed using propositional variables, logical connectives (AND, OR, NOT), and temporal operators. A typical LTL formula might describe properties such as "a certain event eventually happens" or "a certain condition always holds".

Example: Consider the LTL formula

$$\phi = \Box(p \rightarrow \Diamond q) \tag{2.1}$$

which asserts that "it is always the case that if p is true, then eventually q will be true".

This property is valuable in scenarios where you want to ensure that a certain condition (p) implies the eventual occurrence of another condition (q) within the system.

LTL is often used in model checking, a formal verification technique used to check whether a given system model satisfies a desired set of properties. Model checking involves exploring the state space of a system to verify or falsify temporal properties expressed in LTL.

2.1.3 Computation Tree Logic (CTL)

Computation Tree Logic (CTL) (Baier and Katoen 2008) is another temporal logic used in model checking to specify and verify properties of systems. Unlike LTL, which focuses on linear sequences of states, CTL introduces a branching-time perspective, considering all possible paths of a computation simultaneously. This makes CTL suitable for reasoning about systems with non-deterministic behavior.

CTL, similar to LTL, extends propositional logic and includes temporal operators. It is based on the following temporal operators, that can be either quantifier formulas over paths, or path-specific quantifiers formulas, as described below:

- quantifiers over paths:
  - **A**ϕ, where ϕ must to hold on all paths starting from the current state;
  - **E**ϕ, where there exists at least one path starting from the current state such that ϕ holds

- path-specific formulas:
  - **X (Next):** Read as "next state". For example, $\mathbf{X}p$ means "in the next state, p is true".
  - **F (Eventually):** Read as "eventually" or, equivalently, "in the future". For example, $\mathbf{F}p$ means "eventually, p will be true along some path".
  - **G (Always):** Read as "always". For example, $\mathbf{G}p$ means "always, p is true along all paths".
  - **U (Until):** Binary operator denoted as $p\mathbf{U}q$, meaning "until". It indicates that p is true until q becomes true, along some path.
  - **W (Unless):** Read as "unless". For example, $p\mathbf{W}q$, p is true until q becomes true, along some path, but it is possible for q to never be reached. The difference from **U** is that is possible that the condition is never verified.

Example: Consider the CTL formula

$$\phi = \mathbf{AG}(\mathbf{F}\, p \rightarrow \mathbf{F}\, q) \tag{2.2}$$

which asserts that "For all possible paths or states (globally), if p eventually holds along some path, then q will eventually hold along at some point in the future". This property is useful in scenarios where one wants to ensure that a certain condition (p) implies the eventual occurrence of another condition (q) along every possible computation path.

2.1.4 Traffic Light implementation in NuSMV and LTL/CTL property assertion

In this section, a transitional system detailinig a traffic light is used as an example to demonstrate the differences between LTL and CTL.

The example is a traffic light where it transitions from red to green to yellow and back to red. The initial state is red light.

```
MODULE trafficLight (light)

FAIRNESS
    -- Ensure that all processes have fair opportunities to execute
    running;

ASSIGN

next(light) :=
  case
    light = red : green;
    light = yellow : red;
    light = green : yellow;
    TRUE: light;
  esac;

MODULE main

FAIRNESS
    -- Ensure that all processes have fair opportunities to execute
    running;

VAR

  light : {red, yellow, green};

  trafficLight : process trafficLight(light);

ASSIGN

  init(light) := red;
```

Listing 2.1: Traffic Light Example in NuSMV

LTL Properties

The following properties demonstrate how different the verification of the properties for this simple model can be written.

```
LTLSPEC !(light = red) -> (light = green);
```

This LTL property asserts that whenever the light is not red, there exists a path where it must be green. The `!` operator is used for logical negation, and `->` represents logical implication. In this context, it ensures that the system will reach a green light when the red light is off.

```
LTLSPEC (light = red) & !(light = yellow) & !(light = green);
```

This LTL property ensures that if the light is red, it cannot be yellow or green simultaneously. The `&` operator is used for logical conjunction (logic operator AND), ensuring that multiple conditions are true at the same time.

```
1 LTLSPEC (light = red) | (light = yellow) | (light = green);
```

This LTL property asserts that at least one of the three lights—red, yellow, or green—must be on at any given time. The | operator denotes logical disjunction (logical operator OR), meaning that one or more of the conditions can be true. This property ensures that the system always shows a signal.

```
1 LTLSPEC (light = red) xor (light = yellow);
```

This LTL property specifies that either the red light or the yellow light is on, but not both simultaneously. The xor operator is used for logical exclusive XOR, meaning that exactly only one of the conditions must be true. This is important in scenarios where mutually exclusive states are required.

```
1 LTLSPEC !((light = red) xnor (light = green));
```

This LTL property ensures that the lights are in either a red or green state simultaneously or not in either states. The xnor operator denotes logical equivalence (logical operator XNOR), and the ! operator negates it. This property ensures that exists a path that both lights cannot be on or off at the same time (mutually exclusivity and yellow state respectively).

```
1 LTLSPEC (light = red) <-> !(light = green) & !(light = yellow);
```

This LTL property asserts that the red light is on if and only if the green or yellow light is off. The <-> operator represents logical equivalence, meaning both sides of the expression must be either true or false simultaneously. This ensures a direct inverse relationship between the red and green and yellow lights. The state yellow proves that this specification is false.

```
1 LTLSPEC light = green -> X (light = red);
```

This LTL property specifies that if the light is green at the current state, then in the next state (X), the light must be red. The (X) operator denotes the "next" state, so this property is checking that whenever the light is green, it transitions to red in the next step. This enforces a direct sequence of green to red in the model.

```
1 LTLSPEC G ((light = red) -> F (light = green));
```

This LTL property states that globally (G), if the light is red, then eventually (F), the light will turn green. This ensures that whenever the light is red, it will eventually change to green. This property enforces that the system will not stay in the red state indefinitely without transitioning to green.

```
1 LTLSPEC !((light = yellow) U (light = green));
```

This LTL property specifies that it is not the case (!) that the light is yellow until U it turns green. This means that in the model, the light should not remain yellow until it eventually becomes green. It enforces that there should be no continuous period where the light is yellow before it turns green.

```
1 LTLSPEC !((light = red) V (light = green));
```

This LTL property states that it is not the case (!) that the light is red or (V) it is green. The V operator denotes the "releases" operator, which means the light should not be red until it is green, and if it is not green, it should not be red either. Essentially, this property enforces

that either the light must be green or should not be red, ensuring that these conditions are not met simultaneously.

```
1 LTLSPEC light = green -> Y (light = red);
```

This LTL property specifies that if the light is green in the current state, then in the previous state (Y), the light must have been red. The Y operator denotes the "previous" state, meaning this property checks if the light changes from red to green, enforcing that the transition from red to green is sequential and correctly ordered.

```
1 LTLSPEC H (light = red);
```

This LTL property asserts that historically (H), the light has been red at some point in the past. The H operator ensures that the condition (the light being red) has been true at least once before in the history of the system. This property is useful for verifying that the light system has at some point operated in the red state.

```
1 LTLSPEC light = green -> O (light = red);
```

This LTL property specifies that if the light is green in the current state, then in the next state (O), the light must have been red. The O operator denotes the "next" state in the past, meaning it checks the state of the light in the previous step before the current green state. This property ensures that the system transitions from red to green in a valid sequence.

CTL Properties

The following properties demonstrate how different the verification of the properties for this simple model can be written.

```
1 CTLSPEC !(light = red) -> (light = green | light = yellow);
```

This CTL property states that if the light is not red (`!(light = red)`), then the light must be green (`light = green | light = yellow`). The `->` operator represents logical implication. This property enforces that when the light is not in the red state, it should be in the green or yellow state. It ensures that the system adheres to the rule that the light is either of the three states but never any other state when it is not red.

```
1 CTLSPEC (light = red) & !(light = yellow) & !(light = green);
```

This CTL property specifies that when the light is red (`light = red`), it must not be yellow (`!(light = yellow)`) or green (`!(light = green)`). The `&` operator represents logical conjunction. This property ensures that when the light is red, it cannot simultaneously be in any other state, enforcing that red is a distinct state with no overlap.

```
1 CTLSPEC light = red -> (light = red) | (light = yellow) | (light =
    ↪ green);
```

This CTL property asserts that if the light is red (`light = red`), then it must be in one of the defined states: red, yellow, or green (`(light = red) | (light = yellow) | (light = green)`). The `|` operator represents logical disjunction. This property ensures that the light will always be in one of these specific states if it is currently red.

```
1 CTLSPEC (light = red) xor (light = yellow);
```

This CTL property uses the logical exclusive or (`xor`) to state that the light can either be red (`light = red`) or yellow (`light = yellow`), but not both. The `xor` operator ensures that exactly one of these conditions is true at any given time, enforcing mutual exclusivity between the red and yellow states.

```
1 CTLSPEC (light = red) <-> !(light = green) & !(light = yellow);
```

This CTL property represents logical equivalence (`<->`) between the light being red (`light = red`) and the light not being green nor yellow (`!(light = green) & !(light = yellow)`). It ensures that if the light is red, then it must be true that the light is not green nor yellow, and vice versa. This establishes a direct inverse relationship between the red and green and yellow states.

```
1 CTLSPEC light = red -> EX (light = green);
```

This CTL property states that if the light is red (`light = red`), then there exists a next state (`EX`) where the light is green (`light = green`). The `EX` operator denotes "exists next," ensuring that after the light is red, it will eventually turn green in the subsequent state.

```
1 CTLSPEC AG ((light = red) -> AF (light = green));
```

This CTL property states that globally (`AG`), if the light is red (`light = red`), then it will eventually be green (`AF (light = green)`). The `AF` operator denotes "always eventually," ensuring that in all future paths, if the light is red, it will eventually transition to green.

```
1 CTLSPEC EF (light = green);
```

This CTL property specifies that there exists a path (`EF`) where the light will eventually be green (`light = green`). The `EF` operator denotes "exists finally," ensuring that there is some execution path where the light will eventually reach the green state.

```
1 CTLSPEC AG ((light = red) -> AF (light = green));
```

This CTL property is repeated and is similar to the earlier property. It asserts that globally (`AG`), if the light is red (`light = red`), then it will eventually be green (`AF (light = green)`). It ensures that if the light is ever red, it will eventually turn green in all possible future scenarios. This verifies that the transition from red to green is guaranteed across all paths.
More details on how the LTLSPEC/CTLSPEC works is available in the NuSMV user manual[1].

Crucial differences

In this section, the crucial differences between LTLSPECs and CTLSPECs are demonstrated. It is recommended to use LTLSPECs to ensure temporal transitions such as:

```
1     LTLSPEC light = green -> X (light = red);
```

This LTL property ensures that if the light is green in the current state, then in the next state, the light must be red. This is a path-specific property that focuses on the sequence of states directly following the green state. LTL is suitable here because it describes a requirement on the immediate next state of the current path.

[1] Available in https://nusmv.fbk.eu/userman/v26/nusmv.pdf

It is recommended to use CTLSPECs to ensure existential path properties (if a path exists to achieve a certain condition) or universal path properties (all paths reach the desired condition), such as:

```
1    CTLSPEC EF (light = green);
```

This CTL property asserts that there exists some path from the current state where the light will eventually be green. CTL is suitable here because it involves checking whether there is at least one path (from the current state) where a particular condition (light being green) will eventually hold true.

```
1    CTLSPEC AG ((light = red) -> AF (light = green));
```

This CTL property ensures that globally, if the light is red, then it must eventually become green in all possible paths. CTL is appropriate here because it requires a property to hold true across all paths in the computation tree, not just on a single path, thus using the universal quantifier `AG`.
In summary, LTL is ideal for properties that need to be satisfied along individual paths and involve sequential ordering of states. It is best for temporal properties where the exact sequence or timing of state transitions is crucial.
CTL is best for properties that need to be true in all paths or at least one path from a given state, allowing you to specify global constraints or conditions that span multiple possible paths or trees.

2.2 Probabilistic model checking

Probabilistic model checking (Marta Kwiatkowska, Gethin Norman, and David Parker 2017) is a formal verification that analyzes and verifies systems with stochastic or probabilistic behavior. It extends traditional model checking, which is a method for determining whether a given system satisfies a set of desired properties.
In probabilistic model checking, the system's behavior is modeled as a probabilistic or stochastic system in which certain events occur with specific probabilities. This can be useful for analyzing systems that rely on uncertainty, randomness, and/or probabilistic choices to play a significant role, such as in systems involving randomized algorithms, communication protocols, or biological systems.
A classical model for probabilistic model checking is that of discrete-time Markov chains (DTMCs), represented as a tuple D=(S,$\bar{s}$,A,δ,AP,L), where:

- S is a countable set or a state space that represents all possible configurations/states that the modeled system can be in;
- $\bar{s}$ is the initial state of the model and must belong to S;
- A is a finite set of actions;
- δ is a a (partial) probabilistic transition function that maps state–action pairs to probability distributions over S;
- AP is a set of atomic propositions;
- L is a labelling function, containing a set of atomic propositions from a set, that is associated with each state.

In terms of the language for specifying probabilistic properties, a quantitative extensions of temporal logic must be used. Here, it is highlighted a fragment of the logic used as the property specification language for the PRISM model checker, which is known as "PRISM logic". This logic is based on the logics PCTL (probabilistic computation tree logic) and LTL and also incorporates operators to specify expected reward properties.

Two particular probability-related operators are relevant to be described:

- $\mathrm{P}_{\bowtie p}[\varphi]$, specifying that the probability that a path satisfies path formula φ satisfies the bound p;
- $\mathrm{R}^{r}_{\bowtie q}[\rho]$, specifying the expected value of reward formula ρ, under reward structure r, satisfies the bound q;

where a reward structure is a tuple $r = (r_S, r_A)$, where $r_S : S \rightarrow R_{\geq 0}$ is a state reward function and $r_A : (S \times A) \rightarrow R_{\geq 0}$ is an action reward function.
In what concerns temporal operators, PCTL considers a step-bounded until operator besides the classic "next state" and "until" operators, where such step-bounded until evaluates a formula under the semantics of until but limited to a finite number of steps.
Finally, there are reward formulae ρ, for which three operators exist:

- $I^{=k}$ (instantaneous reward): state reward at time step k;
- $C^{\leq k}$ (bounded cumulative reward): reward accumulated over k steps;
- $F\varphi$ (reachability reward): reward accumulated until a state that satisfies φ is reached.

2.3 Model checking in UAVs

In recent years, the application of formal design and verification approaches has gained influence in the context of Unmanned Aerial Vehicles (UAVs). This section undertakes an examination of key contributions within this domain, specifically focusing on the application of formal methods in the design and/or testing of UAVs. The objective is to elucidate how these methods contribute to improving the reliability of UAVs, especially in addressing challenges associated with hazardous environments and communication protocols.

2.3.1 UAV monitoring road conditions in dangerous environment

The authors in (Wang et al. 2021) discuss the challenge of UAVs performing path monitoring tasks in hazardous environments. A method for UAV path selection is proposed, which includes both operator and agent operator modes. The study uses the model checking tool Prism (M. Kwiatkowska, G. Norman, and D. Parker 2011) to model and verify UAV path selection scenes, taking into account scenarios in which the UAV loses contact with the operator in hazardous areas such as mountains or jungles.
In normal conditions, the UAV operates in the operator mode, counting both successful and unsuccessful photographing attempts. In dangerous environments, the agent operator mode is activated, simulating the operator's judgment and allowing the UAV to continue its path monitoring mission. The paper emphasizes the importance of analyzing and verifying UAV path selection in dangerous environments, taking into account different scales and operator characteristics.
The results demonstrate the efficacy of the proposed method in dealing with loss of contact scenarios, providing insights into the mode transition's impact on UAV performance in dangerous environments.

2.3.2 Probabilistic Model Checking and Autonomy

The article described in (Marta Kwiatkowska, Gethin Norman, and David Parker 2022) discusses the increasing importance of ensuring the safe, secure, reliable, timely, and resource-efficient execution of autonomous systems within computing infrastructures, spanning information systems, security, and robotics. Formal modeling and verification, particularly

through probabilistic model checking, are proposed as essential methodologies for designing computer systems.
The challenges posed by autonomous systems operating in uncertain or adversarial environments are addressed, emphasizing the need for controllability and strategic reasoning.
In summary, the unified framework provided by probabilistic model checking, the extension of temporal logic for specifying objectives, and the automatic synthesis of optimal controllers are emphasized. Future issues are highlighted, including the need for practical verification algorithms for partially observable stochastic games, scalability improvements, and addressing the modeling of agents capable of learning and adapting.

2.4 Model checking in protocols

In recent years, the domain of security protocols has witnessed a rising emphasis on formal design and verification approaches to ensure communication system robustness and reliability. This section looks at various major contributions in this field, with a focus on the use of formal approaches in security protocol design.

2.4.1 Secure Group Communication Protocol Verification

The paper by (Hu et al. 1999) evaluates a protocol for secure group communication through the use of finite-state model-checking tools. The protocol involves elements such as a trusted server, authentication mechanisms, and a hierarchical key structure organized in a tree format.
The authors used the Murphi (Rome 2012) model checker and discovered two flaws in the protocol, one of which had already been reported highlighting the importance of formal specification and verification techniques in the field of security protocols. The two flaws in the protocol were the following:

- **Forged Rekey Message Vulnerability:** The bug was previously reported and a solution was created. The Murphi model found that the proposed solution was lacking as it neglected to consider replay attacks. The adversary can save an old rekey message (properly signed by the server) and send it out again later, switching the victim to non-valid keys.
- **Out-of-Order Rekey Messages:** The possibility of out-of-order arrival of rekey messages during multiple transactions lead to potential problems for users in case the messages were correctly decrypted. The authors proposed two solutions: for a small network with rare rekey messages, users could save unintelligible messages and retry them upon new rekey messages, but this was deemed impractical for frequent rekeying; another approach involves implementing a scheme for order-preserving broadcast on a non-order-preserving network but it comes with substantial overhead and scalability issues for larger networks.

The paper also discusses the challenges encountered during the modelling and model-checking processes and focuses its attention on the effectiveness of model checking in identifying vulnerabilities. However, it suggests that while model checking is effective, it should act as complement to conventional proof-based methodologies for critical applications, rather than replacing them. The paper concludes with proposed fixes for the protocol and recommendations for enhancing verification tools.

2.4.2 Model Checking in Large Network Protocol Implementations

The work described in (Musuvathi and Engler 2004) addresses the challenges associated with testing large network protocol implementations. Conventional testing methods are frequently insufficient for investigating the state space of these systems. The paper introduces model checking as a formal verification technique, highlighting its use in real-world, well verified protocol implementations.
It discusses techniques used in CMC (Musuvathi, Park, et al. 2002), a C model checker, with an emphasis on their application to the Linux TCP/IP implementation. The authors successfully identified four problems in the protocol implementation, illustrating model checking scalability in the context of complex systems.
The four problems are presented as the following:

- **Memory Errors:** The authors detected three types of memory errors - not checking for allocation failure (12 errors), not freeing allocated memory (8 Errors) and using memory after freeing it (2 errors).
- **Unexpected Messages:** They also detected two instances where unexpected messages caused the system to crash with a segmentation violation.
- **Invalid Messages:** The authors detected four cases of invalid packets being created, two cases of using uninitialized variables and two cases where invalid routes were used to send routing updates, violating the Ad Hoc On Demand Distance Vector (AODV) specification.
- **Routing Loops:** They also found three routing loops, two caused by implementation errors and one due to a bug in the AODV protocol specification.

These problems highlight issues with memory management, message handling, specification adherence, and routing logic in AODV implementations. The research argues that further improvements can be made with a more thorough reference model, as well as future initiatives that involve running two TCP implementations concurrently to cross-check their behavior.

2.4.3 Comprehensive Approach to Critical Avionics Systems Verification

The paper by (Elkholy et al. 2020) presents an approach to address challenges in modeling, verifying, and testing intelligent critical avionics systems.
The formalism of extended interpreted systems is introduced to support intelligence, autonomy, communication, and conditions. The authors use model checking and a testing methodology to demonstrate efficiency and scalability in tests. Despite positive results, the paper acknowledges limitations due to timing constraints and the lack of probabilistic considerations in critical avionics systems.
Future work is planned to extend the concept to intelligent cyber-physical systems, incorporating timing constraints and probabilistic aspects, and utilizing deep learning for dynamic behavior adaptation.

2.5 Tools

Model checking relies on a wide spectrum of tools designed to cater to specific facets of system verification, playing a pivotal role in ensuring the correctness and dependability of a diverse array of systems. In this section, we briefly introduce notable tools employed in the field and which we understand as candidates to be used in the development of the present dissertation.

2.5.1 SPIN (Simple Promela Interpreter)

The Simple Promela Interpreter (SPIN) (Authors 2020) is one of the most well-known model checkers that have been used in the verification of real-time problems, and its focus is primarily for the verification of concurrent systems. SPIN was written by Gerard J. Holzmann and coleagues from the original Unix group of the Computing Sciences Research Center at Bell Labs, beginning in 1980.
The usage of SPIN for verification purposes is performed in combination with the Promela modeling language which has been designed to support the modeling of asynchronous distributed algorithms as Buchi automata, whereas the properties to be verified assume the form of LTL formulae. An example of a simple specification written in the Promela language is presented in Figure 2.2.
Besides acting as the model checker for Promela models, SPIN can also act as a simulator, allowing for following a path of execution of the target system and presenting the outcomes of that simulation to the user. This is often a very important aspect of model checking in the sense that it helps developers fine tuning the identification of problems with both the model and specification.

```
bool turn, flag[2]; // shared variables
byte ncrit;         // number of processes in critical section

active [2] proctype user() // two processes
{
 assert(_pid == 0 || _pid == 1);
again:
 flag[_pid] = 1;
 turn = _pid;
 (flag[1 - _pid] == 0 || turn == 1 - _pid);

 ncrit++;
 assert(ncrit == 1); // critical section
 ncrit--;

 flag[_pid] = 0;
 goto again
}
```

Figure 2.2: Simple example of a Promela model.

2.5.2 NuSMV (New Symbolic Model Verifier)

Widely used for the formal verification of hardware and software systems, NuSMV (*NuSMV: A Symbolic Model Checker* n.d.) is a symbolic model checker that operates on transition systems. It allows the verification of properties specified in temporal logics such as LTL and CTL, and Real-Time CTL. Moreover, it is also extensible and supports various input languages.

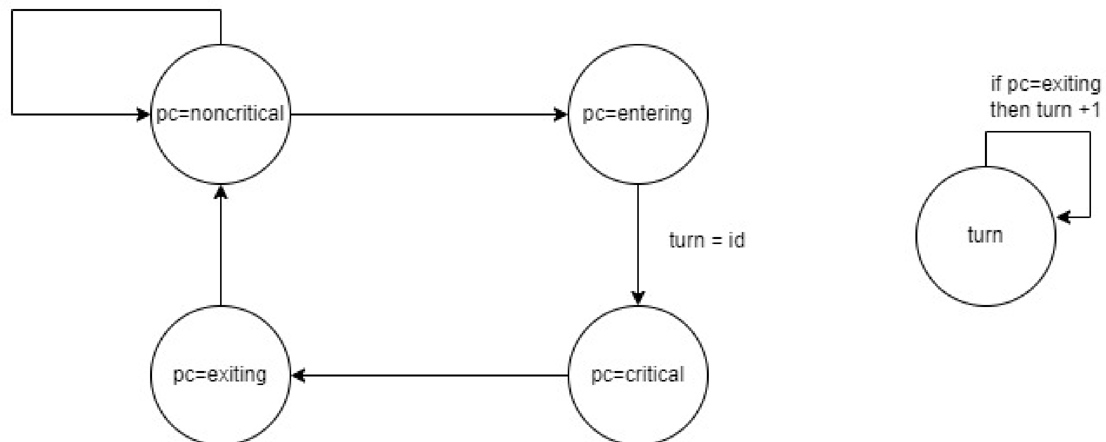


Figure 2.3: NuSMV visual representation of the code example

```
1  -- First, we create the module of the process
2  MODULE P(id, turn)
3  VAR
4      pc : {entering, noncritical, critical, exiting};
5  ASSIGN
6      -- We enforce that a process never starts in a critical section
7      init(pc) := noncritical;
8      -- We then define the transition relation
9      next(pc) :=
10         case
11             -- If the process is in a noncritical section, then
                ↪ either remains as is or tries to enter the critical
                ↪ section
12             pc=noncritical : {noncritical,entering};
13             -- If the process is trying to enter a critical section
                ↪ , and his its turn, then the process enters the
                ↪ critical section
14             pc=entering & turn=id : critical;
15             -- If the process is already in the critical section,
                ↪ then he  tries exiting it
16             pc=critical : exiting;
17             -- If the process is already exiting the critical
                ↪ session, then it moves to a noncritical section
18             pc=exiting : noncritical;
19             -- Otherwise, stay as is
20             TRUE : pc;
21         esac;
22     -- Determine how turn evolves
23     next(turn) :=
24         case
25             -- If the process is exiting a critical section, then
                ↪ update the turn
26             pc=exiting : (turn+1) mod 2;
27             -- Otherwise, stay as is
28             TRUE : turn;
29         esac;
30 -- We impose that processes are fair, that is, that both are given
                ↪ similar oportunities to run
31 FAIRNESS running
32 -- Main module: simply instantiate both processes and run them
33 MODULE main
34 VAR
35     p0 : process P(0, turn);
36     p1 : process P(1, turn);
37     -- We define the turn variable, but do not impose constraints
                ↪ over it.
38     turn : 0..1;
39
40 LTLSPEC G !(p0.pc = critical & p1.pc = critical)
41 LTLSPEC G (p0.pc = entering -> F p0.pc = critical)
```

Figure 2.4: Example of NuSMV code (Pereira and Bragança 2022)

The code present in figure 2.4 defines a model of two processes, **P0** and **P1**, that share a single resource (*turn*) using a "turn-based" mutual exclusion algorithm.
The code defines a module for the process, which has two parameters: *id* and *turn*. The *id* parameter is used to distinguish between the two processes, and the *turn* parameter is used to track whose turn it is to enter the critical section.
The process module has a variable, *pc*, which represents the current state of the process. The possible values of *pc* are **entering**, **noncritical**, **critical**, and **exiting**. The initial value of *pc* is **noncritical**, ensuring that the process never starts in the critical section.
The transition relation for the process is defined in the function **next(***pc***)**, which specifies how the value of *pc* changes based on its current value and the value of the *turn* variable.

- The process can remain in the **noncritical** section or try to enter the critical section.
- If it is trying to enter the critical section and it is its turn, it enters the critical section.
- If it is already in the critical section, it tries to exit.
- If it is already exiting, it moves to the **noncritical** section.

The *turn* variable is updated in the function **next(***turn***)** of the process module as well. If the process is exiting, the *turn* variable is updated by taking the current value and adding 1 module 2, giving turn to the other process.
The **FAIRNESS** statement in the code ensures that both processes are given fair opportunities to run.
The main module instantiates both processes, **P0** and **P1**, and defines the turn variable. The turn variable is not constrained in any way, allowing the processes to take turns accessing the critical section.
The LTLSPEC statements define properties of the system that should always hold. The first LTLSPEC specifies that the two processes should never be in the critical section at the same time, ensuring mutual exclusion. It can be read as "It is always the case that processes **P0** and **P1** cannot be in the critical section at the same time.
The second LTLSPEC specifies that if **P0** is entering the critical section, it will eventually enter the critical section, ensuring progress. It can be read as "Globally, if process **P0** is **entering** the critical section, then it will eventually enter the **critical** section".
After running the code, the first LTLSPEC is confirmed as true while the second LTLSPEC is confirmed as false and presented with a counter-example. The counter-example contains a step-by-step demonstration on how the LTLSPEC can fail.
Overall, the code presented in figure 2.4 defines a model of a turn-based mutual exclusion algorithm and uses formal verification techniques (LTLSPECs) to ensure that it satisfies the desired properties.

2.5.3 PRISM

PRISM (M. Kwiatkowska, G. Norman, and D. Parker 2011) is a probabilistic model checker that is used to formalize and analyze systems that exhibit random or probabilistic behavior. Its applications are wide, encompassing communication protocols, distributed algorithms, security, and biological systems. The tool supports a wide range of probabilistic models, including discrete-time and continuous-time Markov chains, as well as Markov decision processes.
Models described in the PRISM language are examined for quantitative properties using temporal logics such as Probablistic Computation Tree Logic (PCTL), Continuous Stochastic Logic (CSL), Linear Temporal Logic (LTL), and Probablistic Computation Tree Logic Extended (PCTL*).

The tool uses symbolic data structures and algorithms, such as Binary Decision Diagrams (BDDs) and Multi-Terminal Binary Decision Diagrams (MTBDDs), and supports a variety of analytical approaches, including abstraction refinement and symmetry reduction.

2.5.4 UPPAAL

UPPAAL (Pettersson and Larsen 2000) is an integrated tool environment designed for modeling, simulation, and verification of real-time systems. Suited for systems with non-deterministic processes, finite control structures, and real-valued clocks, UPPAAL finds applications in areas such as real-time controllers and communication protocols.
The tool comprises a description language, simulator, and model-checker, which together facilitate the tasks of modeling, validation, and verification of system behavior.
Notable features include a graphical system editor, a graphical simulator for dynamic behavior visualization, and a model-checker for automatic verification of safety properties.
In figure 2.5, it is possible to observe an example of a coin flip model provided by the tool itself.

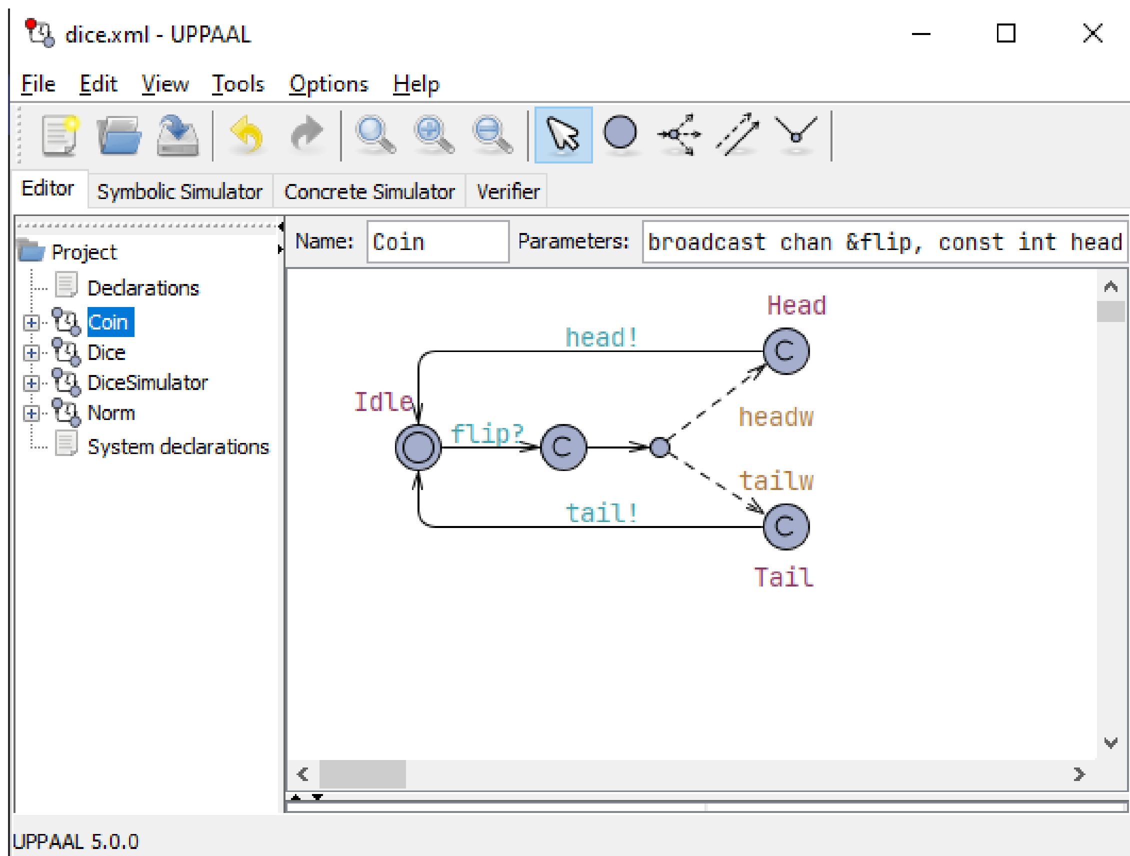


Figure 2.5: UPPAAL coin example available as an example template in the program

The initial state is **Idle**. When the flip occurs it shall either land on **Head** or **Tail**. When the result is revealed, it shall transition into **Idle** until next coin flip.

Chapter 3

Analysis

This chapter will analyse the various aspects of the overall functionality of the authorization transfer protocol presented in (Ferreira 2023). Furthermore, we present the initial efforts for the definition of the scenario that will be subject to a formal specification and model checking tasks to be further developed to reach the objectives of the thesis. These efforts correspond to an initial analysis of the scenario, its actors, and the derivation of an initial set of requirements which will be core for developing formal specification to be model checked against the model to be developed.
The author in (Ferreira 2023) did not implement several features however in consideration for the protocol as a whole, these features will be presumed to be implemented for the following sections.

3.1 Scenario Definition

According to the authors (Ferreira 2023), the system is defined by two ground control units and the UAV. The scenario in figure 3.1 is defined as "must be able to deliver a packet crossing the authority handover region and turn back to the launch".
The UAV shall be launched from a point in the coverage area GCS-A. Within the coverage area of GCS-A, it shall be controlled by ground control station A and it shall be controlled by ground control station B, in the coverage area GCS-B. There is an area named Authority Handover Region which is the intersection between both coverage areas. In the Authority Handover Region, it is possible to transfer control over the UAV from one ground control station to another ground control in range. This scenario can be extended to consider

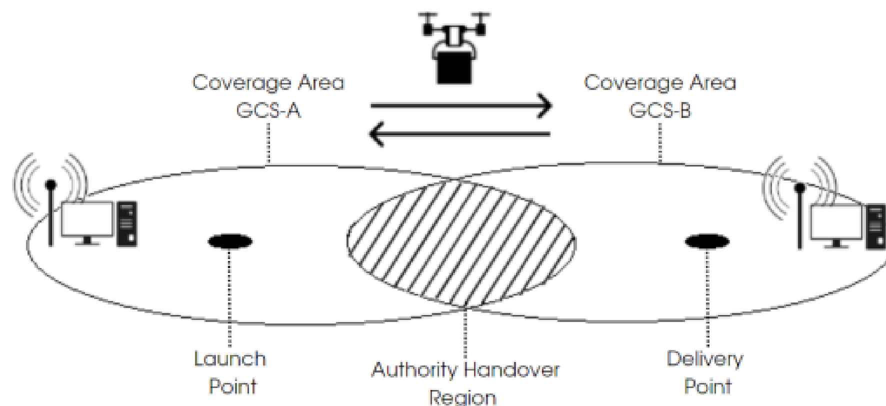


Figure 3.1: Authority Handover Procedure available in (Ferreira 2023)

multiple ground controls, however the focus will be in a scenario where only two ground controls exists.
The system architecture will be simplified by assuming that the UAV and each ground control will be a component, and not seeing the system as an agglomeration of components. The UAV shall be controlled by one ground control unit at a time.
The following sections will describe the phases of the communication protocol, including start-up, mission execution, handover and the ground control. The start-up is defined by actions to be taken during system initialization, the mission execution specifies the procedures to be followed during the execution of the mission, the handover details the steps involved in the process of the control handover and the ground control defines the interactions with the UAV. It will be followed by the derived requirements of these sections, that will be used as a baseline for the execution of the thesis.

3.2 On Start-up

In this section we address the procedure that represents how the UAV operates at system level during start-up. A visual representation of the procedure is presented in Figure 3.2. The figure is abstracted to a functional level in order to simplify the complexity of the procedure. On start-up of the UAV, the drone parameters shall be loaded from configuration and three tasks shall be initiated and executed in parallel. All tasks, which require connection to the ground control to proceed, are the following:

- **Task 1** - Connect to the ground control and get telemetry.
- **Task 2** - Connect to the ground control, display current zone information. The current zone information shall be continuously transmitted to ground control until an exit command is received and the connection is afterwards closed.
- **Task 3** - Connect to the ground control, verify if the connection is active and continuously check for incoming instruction or if the connection is lost. If an instruction is received, it shall be executed and after execution, it shall continuously check for incoming instruction and server connection.

3.3 On Mission

In this section we address the procedure that specifies how the UAV operates on mission. In "On mission" state, the UAV is listening for messages. Every message shall be verified if it's a known message type, pass through a security layer and verify if it originates from a authorized GCS zone. In case it is not a known message type or originates from a non authorized GCS zone, then the message is promptly ignored and shall reset back to listening for new messages.
In case the UAV is in mission state, the drone shall be responsible to load the mission details and setting up the handover. All tasks, which require connection to the ground control to proceed, are the following:

- **Task 1** - Load mission details from ground control.
- **Task 2** - Verify geofences and periodically check distance to delivery point.
- **Task 3** - Start handover procedure.

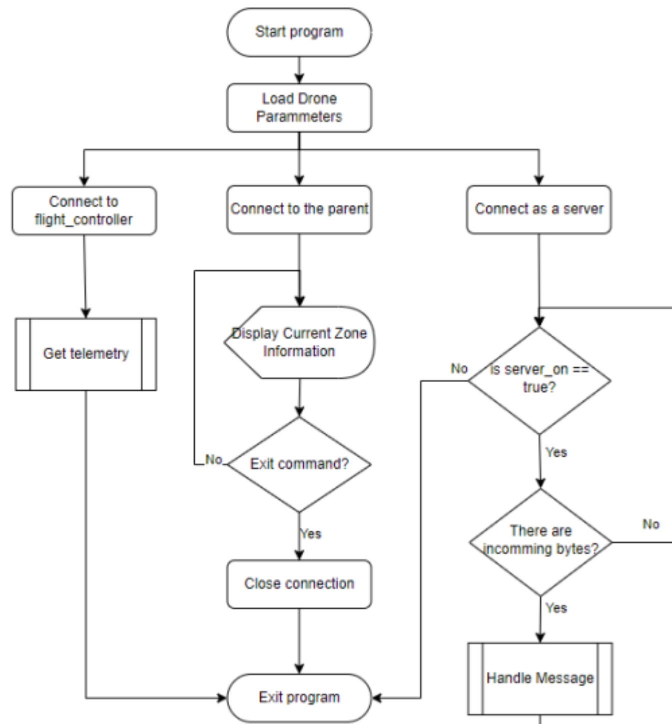


Figure 3.2: Procedure available in (Ferreira 2023)

3.4 On Handover

The following procedure represents how the UAV executes the handover procedure. In "On handover" state, the UAV sends telemetry information to the current ground controller, and waits for verification from the ground controller in charge. If the UAV system detects that it is in a handover region, it shall connect to the second ground control with authorization from the current ground controller in charge. All tasks, which require connection to the ground control to proceed, are the following:

- **Task 1** - Send telemetry to ground control in charge periodically.
- **Task 2** - Verify if it's handover region and start handover procedure.
- **Task 3** - Communication with the ground control in charge. Exchange of information, connection authorized, connection accepted/refused.
- **Task 4** - Communication with the partnered ground control. Exchange of information, connection request, connection accepted/refused.
- **Task 5** - Change ground control in charge.

The full sequence is represented in figure 3.3.

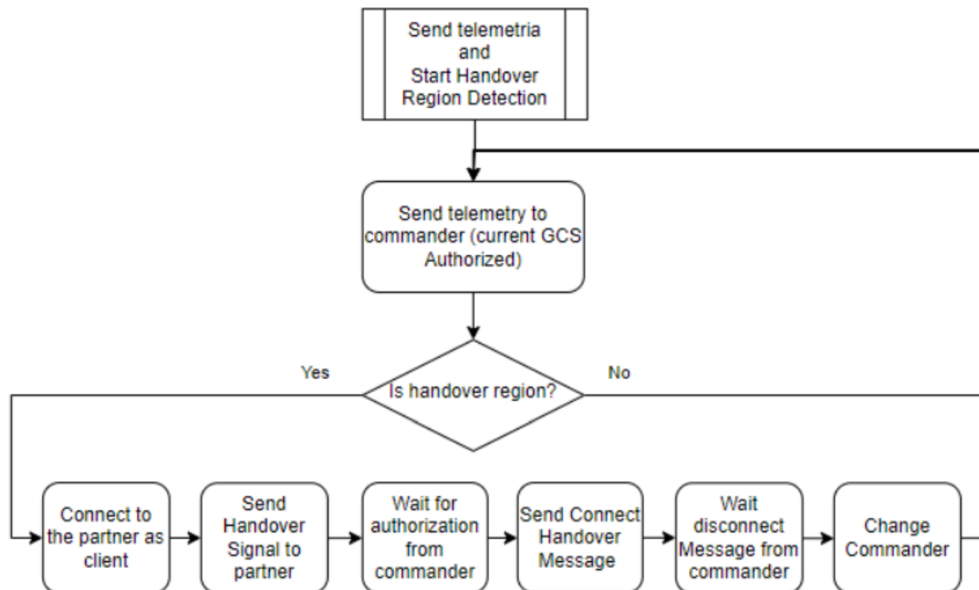


Figure 3.3: Handover Procedure available in (Ferreira 2023)

3.5 Ground Control

The procedure, present in figure 3.4, represents how the ground control main function operates. The ground control shall load the configured parameters and start-up the server. The server shall have a parallel task handling messages. The main function shall load the mission and await the input commands from the keyboard. It shall handle the commands from the keyboard and if it's an exit command, it shall shutdown the server, close the connection and finish the program.
The ground control shall be responsible to load the mission details, exchange information with the UAV and setting up the handover. All tasks, which require connection to the UAV to proceed, are the following:

- **Task 1** - Transmit mission details to the UAV.
- **Task 2** - Receive and process telemetry from UAV.
- **Task 3** - Start handover procedure as the ground control in charge.
- **Task 4** - Start handover procedure as the partnered ground control.
- **Task 5** - Process keyboard input and transmit information to the UAV.

The ground control command handling, represented in figure 3.5, is based on numeric input:

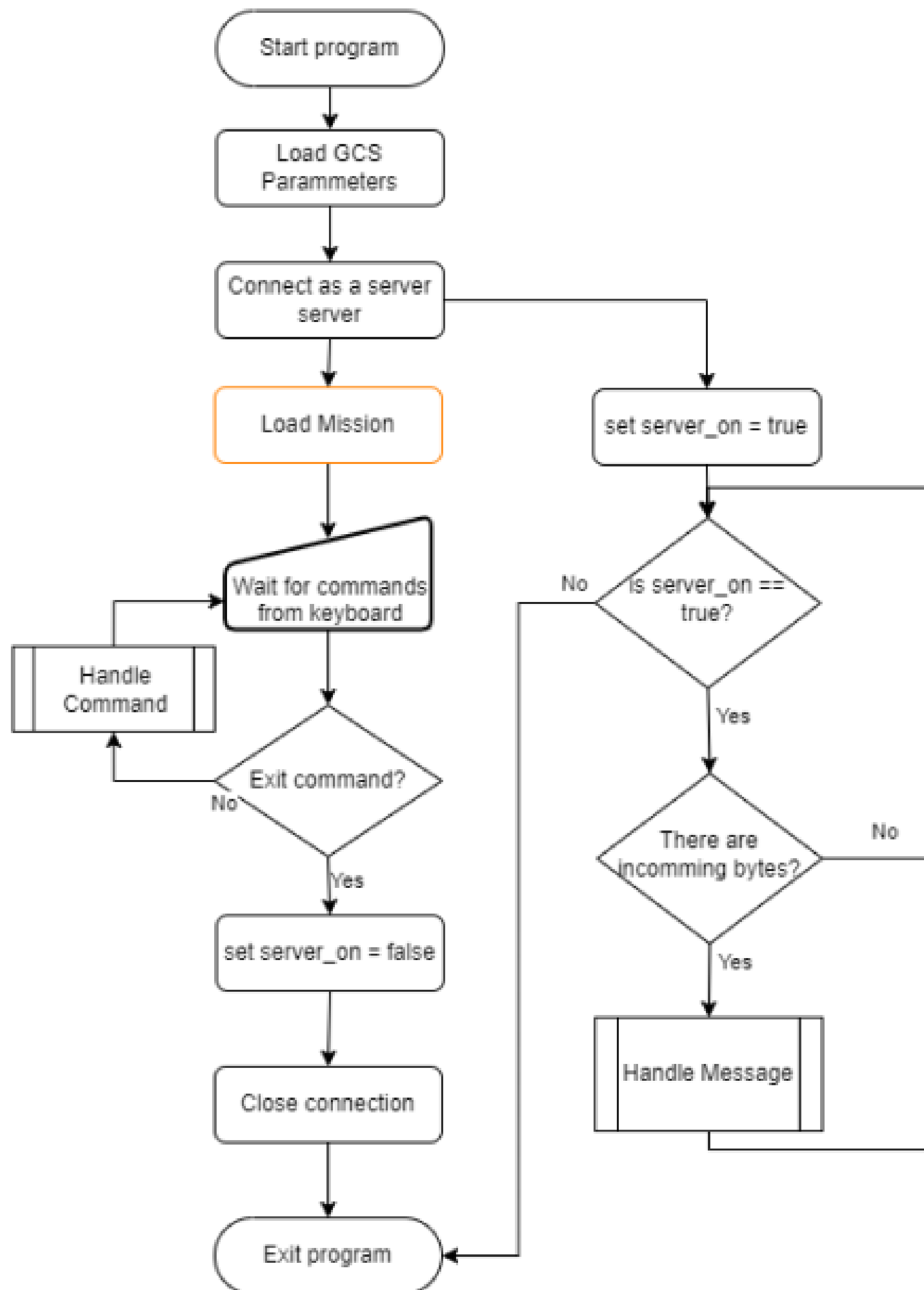


Figure 3.4: Ground Control Procedure available in (Ferreira 2023)

- 0 - Exit program.
- 1 - Share Mission.
- 2 - Send Start Command.
- 3 - Send Stop Command.
- 4 - Send Continue Command.

If the command type is not recognized, it is promptly ignored.

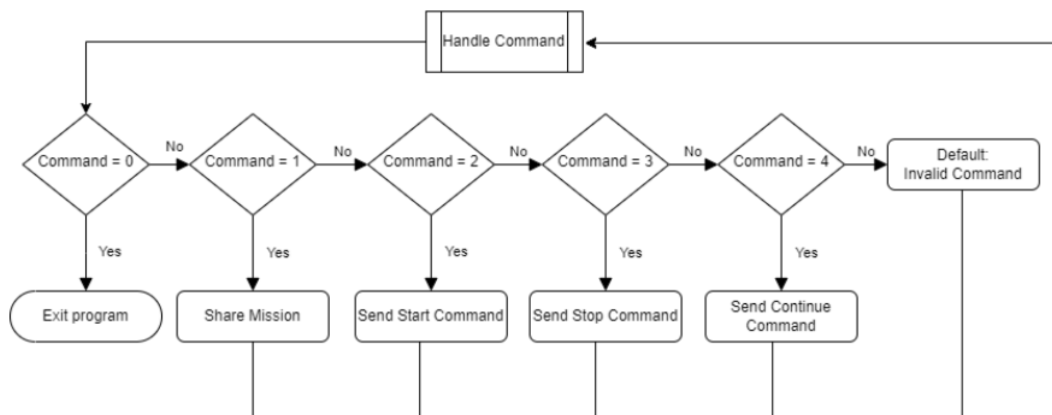


Figure 3.5: Ground Control Commands available in (Ferreira 2023)

The procedure presented in figure 3.6 represents how the ground control handles messages. There are seven types of messages that can be obtained:

- **TELEMETRY** - Displays telemetry and verifies if the UAV is on route.
- **CONNECT** - Sets up drone connection and sends acknowledgement message to drone.
- **MISSION** - Load mission and sends drone connection
- **HANDOVER_REQUEST** - Send On_Handover message to the second ground control.
- **HANDOVER** - Send OK_Handover message to the drone.
- **CONNECT_HANDOVER** - Starts drone communication as client and sends DRONE_CONNECTED to the ground control.
- **DRONE_CONNECTED** - Resets the check route option and sends OK_DISCONNECT_HANDOVER to the drone.

In case the message type is not any of the previously stated types, it shall be promptly ignored.

The figure 3.7 represents the temporal diagram which will function as a base model for the analysis.

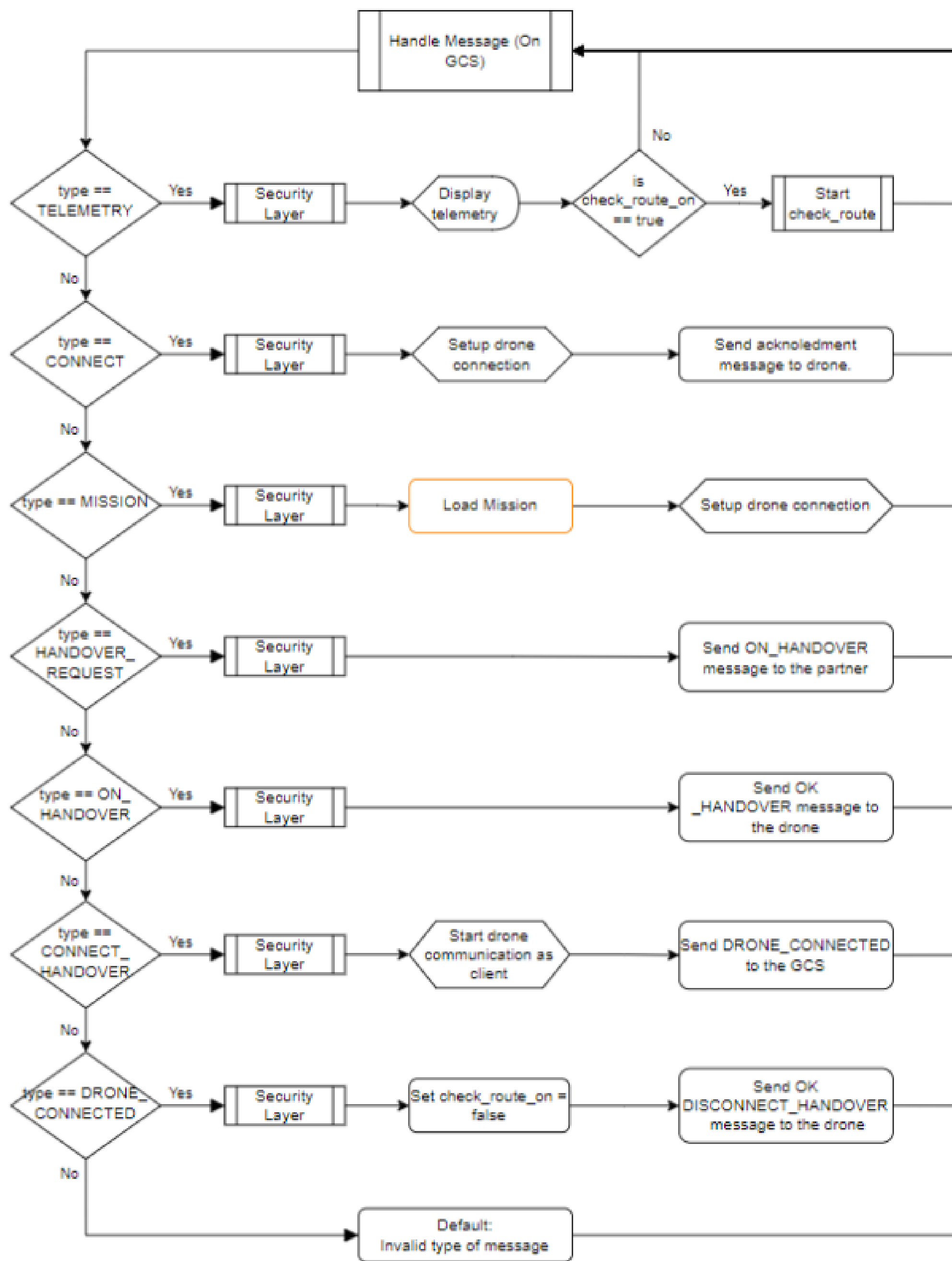


Figure 3.6: Ground Control Messages available in (Ferreira 2023)

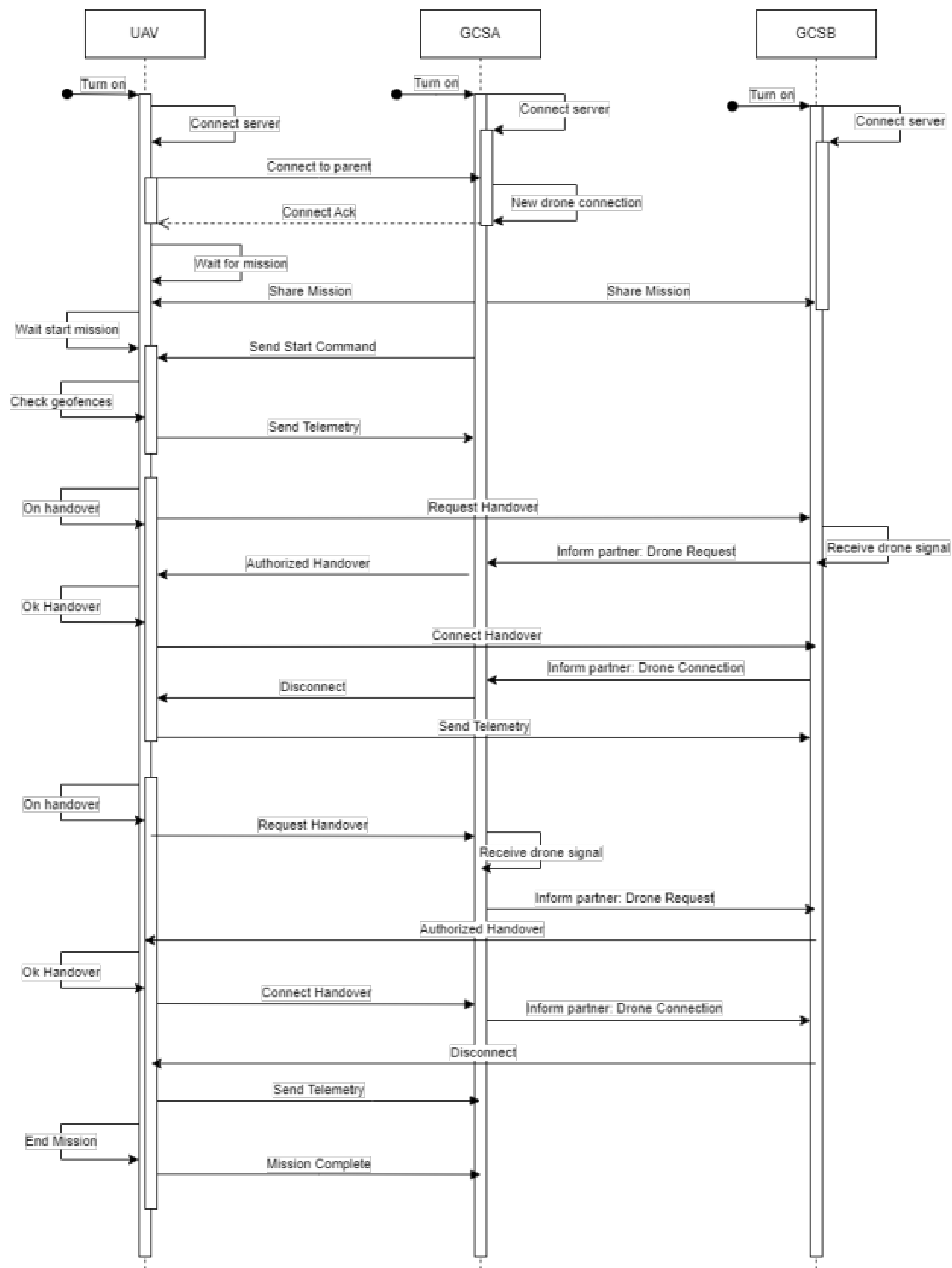


Figure 3.7: Temporal diagram of the handover procedure available in (Ferreira 2023)

3.6 Derived Requirements

Based on the information presented in the previous subsections, where each of them describe the procedures for the various mission stages of a UAV, the next logical step is to derive the requirements that will form the baseline for all the formal modelling, specification, and verification that will be addressed during the development of the thesis. It is however important to note that these requirements may change during the evaluation. Moreover, and to reduce the potential for ambiguity in the derivation of the requirements, we will adopt the Easy Approach to Requirements Syntax (EARS) approach to requirement specification, a well-know technique for requirement specification in the industry, notably in the aerospatial domain[1].

3.6.1 UAV Requirements

- **REQ-UAV-001** - The UAV shall proceed to state "Start-up" when powered on.
- **REQ-UAV-002** - If the UAV is in the "Start-up" state and is powered off, then it shall transition to "Power Off" state.
- **REQ-UAV-003** - If the UAV is in the "Operating" state and is powered off, then it shall transition to "Power Off" state.

Start-up

- **REQ-UAV-004** - When the UAV transitions to "Start-up" state, it shall load up the UAV parameters from the configuration data.
- **REQ-UAV-005** - When UAV is in "Start-up" state and has finished loading UAV parameters from configuration data, it shall connect to the configured Ground Control Station.
- **REQ-UAV-006** - When the UAV connects to the configured Ground Control Station, it shall transition to "Operating" state.

Operating - Telemetry Information

- **REQ-UAV-007** - When the UAV transitions to "Operating" state, it shall connect to the flight controller. In addition, it shall keep retrying the connection request until it is successful.
- **REQ-UAV-008** - When the UAV is in "Operating" state and connected to the flight controller, it shall get current telemetry and transmit it to the ground control station.

Operating - Message Handling

- **REQ-UAV-009** - While the UAV transitions to "Operating" state, it shall listen continuously for incoming messages from ground control stations.
- **REQ-UAV-010** - While the UAV is in "Operating" state, it shall validate any incoming messages from ground control stations.
- **REQ-UAV-011** - While the UAV is in "Operating" state, it shall discard any invalid incoming messages from ground control stations.
- **REQ-UAV-012** - While the UAV is in "Operating" state, if the message type is "Keyboard Command", it shall perform a movement maneuver according to the content of the message.

[1] EARS - Easy Approach to Requirements Syntax - https://alistairmavin.com/ears/

- **REQ-UAV-013** - While the UAV is in "Operating" state, if the message type is "Mission", it shall verify the geofences.
- **REQ-UAV-014** - While the UAV is in "Operating" state, if the message type is "Mission", it shall transmit distance to goal.
- **REQ-UAV-015** - While the UAV is in "Operating" state, if the message type is "Handover - Connect to Partner as Client" and the UAV is within handover region, it shall send a message request to connect to partner. In addition, it shall retry to transmit the message in case of failure.
- **REQ-UAV-016** - While the UAV is in "Operating" state, after successfully transmitting the message request to connect to partner, it shall transmit a handover request to partner. In addition, it shall retry to transmit the message in case of failure.
- **REQ-UAV-017** - While the UAV is in "Operating" state, after successfully transmitting the connect handover request to partner and disconnect message from commander, it shall change commander to current partner.

3.6.2 Ground Control

- **REQ-GCS-001** - The GCS shall proceed to state "Start-up" when powered on.
- **REQ-GCS-002** - If the GCS is in the "Start-up" state and is powered off, then it shall transition to "Power Off" state.
- **REQ-GCS-003** - If the GCS is in the "Operating" state and is powered off, then it shall transition to "Power Off" state.

Ground Control - Start-up

- **REQ-GCS-004** - When the GCS transitions to "Start-up" state, it shall load up the parameters from the configuration data.
- **REQ-GCS-005** - When the GCS is in "Start-up" state and has loaded configuration data, it shall connect to the configured UAV.
- **REQ-GCS-006** - When the GCS has connected to the configured UAV, it shall transition to "Operating" state.

Ground Control - Message Handling

- **REQ-GCS-007** - When the GCS transitions to "Operating" state, it shall continuously listen to incoming messages.
- **REQ-GCS-008** - While the GCS is in "Operating" state, it shall validate any incoming messages from the UAV.
- **REQ-GCS-009** - While the GCS is in "Operating" state, it shall discard any invalid incoming messages from the UAV.
- **REQ-GCS-010** - While the GCS is in "Operating" state and received message type is "Telemetry", it shall display latest telemetry data.
- **REQ-GCS-011** - While the GCS is in "Operating" state and "check route" is configured, it shall analyse latest telemetry to verify if the UAV is currently on route.
- **REQ-GCS-012** - While the GCS is in "Operating" state and received message type is "Connect", it shall setup the UAV connection and transmit an acknowledgement message to the UAV upon a successful connection setup.
- **REQ-GCS-013** - While the GCS is in "Operating" state and received message type is "Mission", it shall load distance to mission goal from UAV.

- **REQ-GCS-014** - While the GCS is in "Operating" state and received message type is "Handover Request", it shall transmit an acknowledgement message to the UAV.
- **REQ-GCS-015** - While the GCS is in "Operating" state and received message type is "On Handover", it shall transmit an "OK Handover" to the UAV.
- **REQ-GCS-016** - While the GCS is in "Operating" state and received message type is "Connect Handover", it shall set UAV communication as client and transmit a "UAV Connected" to the UAV.
- **REQ-GCS-017** - While the GCS Commander is in "Operating" state and GCS Partner is connected to UAV, the commander shall disconnect and the partner shall be the new commander.

Ground Control - Command Handling

- **REQ-GCS-018** - When the GCS is in "Operating" state and it detects a keyboard command, it shall transmit a validated command to the UAV.

3.7 Conclusions

This chapter analysed the various aspects of the overall functionality of the authorization transfer protocol presented in (Ferreira 2023). A scenario definition was established as the foundation for subsequent analysis. The key procedures and actions of the protocol's behaviour during start-up, mission execution and handover were identified. The ground control role, responsibilities and communication handling were presented as well.
Requirements were derived from the performed analysis, functioning as the initial baseline for future iterations of the requirements.
This holistic approach contributes to the protocol's robustness, adaptability, and effectiveness in the specified operational context.

Chapter 4

Methodology

This chapter describes the research and evaluation methodology proposed to accomplish the thesis objectives.

4.1 Research approach

This thesis is an applied research effort focused on advancing the development of techniques and procedures enabling a drone to execute an authority handover from one ground control to another.
Firstly, there will be a comprehensive literature review to assimilate information and categorize related works. This involves an in-depth exploration of existing research papers, articles, and other academic resources that elucidate the application of model checking in the context of UAV communication protocols.
Following the literature review, the research approach will encompass the following phases:

- **Requirements Definition:** Clearly define the requirements for the UAV authority handover protocol. This involves understanding the operational scenarios, constraints, and key functionalities.
- **Model Definition in Selected Tool:** Implement the defined requirements in a selected model checking tool. This phase involves translating the conceptual requirements into a formalized model for analysis.
- **Analysis of Results:** Execute the model in the selected tool and analyze the results. This involves assessing the protocol's behavior under different scenarios and identifying any discrepancies or potential issues.
- **Iteration of Requirements, Model, and Results:** Based on the analysis, iterate on the requirements, refine the model, and reanalyze the results. This iterative process ensures the continuous improvement of the protocol.
- **Results Analysis and Discussion:** Provide a comprehensive analysis of the results and engage in a discussion that interprets the implications and significance of the findings.

4.2 Evaluation

The evaluation phase involves assessing the effectiveness and efficiency of the UAV authority handover protocol. This includes:

- **Protocol Performance:** Evaluate how well the protocol facilitates the authority handover process, considering factors like reliability and adaptability.
- **Tool Performance:** Assess the performance of the selected model checking tool in accurately representing and analyzing the UAV communication protocol.
- **Scalability:** Explore the scalability of the protocol and the model checking approach concerning the size and complexity of communication networks.

Chapter 5

Model Checking Approach

In this section, the iteration of the models are presented. Each iteration is composed of the NuSMV model with the LTL/CTL properties related to the derived requirements. At the end of each iteration, a retrospective is done as well as what should change in the following iteration.

5.1 First Iteration of the model

The model created in NuSMV is based on the requirements defined in section 3.6, where the requirements were derived from the proposed communication protocol. In this section, three NuSMV modules, which compose the entire NuSMV model, will be analysed:

- **Module uavControl:** Responsible for the UAV actions
- **Module GcsControl:** Responsible for both GCSs actions.
- **Module main:** Oversees the communication as a third party and introduces elemental external to the UAV and GCS.

5.1.1 Unmanned aerial vehicle (UAV) module

For this module, the requirements REQ-UAV-(001..017) are implemented in this module.

```
1 MODULE UavControl (commanderId, stateUav,
             ↪ varUav_connectionToGcsCommander,
             ↪ varGcs_isCommanderConnectionAuthorized,
             ↪ varUav_messageTypeToTransmitToCommander,
             ↪ varUav_messageTypeToTransmitToPartner,
             ↪ varUav_messageTypeToReceiveFromCommander,
             ↪ varUav_messageTypeToReceiveFromPartner,
             ↪ varUav_checkDistanceToGoal, commanderHasSwitched,
             ↪ keyboardPress)
```

Figure 5.1: Module uavControl definition

Each variable defined for the **Module uavControl** is used as an interface, which means the **Module main** can define variables that can be inserted in the module. This can be viewed as the **Module uavControl** is a subfunction of the **Module main**. The definition of each variable is the following:

- **commanderId**: Defines the ID of the GCS Commander. Scale/Interpretation: 1 = GCS A, 2 = GCS B;

- **stateUAV**: poweredOff, startUp, operational; – Possible states for the UAV system. Scale/Interpretation: poweredOff = UAV is powered off, startUp = UAV is in start-up procedure, operational = UAV is in operational state;
- **varUav_connectionToGcsCommander**: Defines that the UAV is connected to a GCS (commander). Scale/interpretation: noConnection = No connection available; attemptingToConnect = Has requested connection to the GCS, connected = Is currently connected to the GCS commander;
- **varGcs_isCommanderConnectionAuthorized**: Defines the authroization for a start-up connection for the UAV to connect to the GCS commander. Scale/Interpretation: TRUE = Authorized connection, FALSE = Not authorized;
- **varUav_messageTypeToTransmitToCommander**: none, telemetry, mission, onHandover; – Defines the possible message types to send from UAV to the GCS. Scale/Interpretation: none = No message available, telemetry = Telemetry type message, mission = Mission type message, onHandover = Proceeding with handover procedure;
- **varUav_messageTypeToTransmitToPartner**: Defines the possible message types to send from UAV to the GCS. Scale/Interpretation: none = No message available, handoverRequest = Request handover to GCS Partner, connectHandover = Connect to GCS Partner;
- **varUav_messageTypeToReceiveFromCommander**: Defines the possible message types to send from UAV to the GCS. Scale/Intepretation: none = No message available, telemetry = Telemetry request message, mission = Mission message type, okHandover = Authorize handover to connected partner, keyboardCommand = Keyboard type command to perform movement;
- **varUav_messageTypeToReceiveFromPartner**: Defines the possible message types to send from UAV to the GCS. Scale/Interpretation: none = No message available, acknowledgeConnectMessage = Accept connect message;
- **varUav_checkDistanceToGoal**: Defines the distance to goal as an integer. Scale/interpretation: 1 = 1 meter; Range = 0..99 meters;
- **commanderHasSwitched**: Defines if the commander has switched. Scale/Interpretation: TRUE = Commander has switched during the current handover procedure, FALSE = Commander has not switched during the current handover procedure;
- **keyboardPress**: Possible directions for the keyboard presses. Scale/Interpretation: none = No key is being pressed, down = Downwards movement, up = Upwards movement, left = Left movement, right = Right movement;

Initialization

The module is used to create a process in the **Module main** with variables created in the **Module main**.
For the initialization, the initial value for the variables are the following:

- **commanderId** = 1;
- **stateUAV** = poweredOff;
- **varUav_connectionToGcsCommander** = noConnection;
- **varGcs_isCommanderConnectionAuthorized** = FALSE;
- **messageFromUavToGcsCommander** = none;
- **messageFromUavToGcsPartner** = none;
- **messageFromGcsCommanderToUav** = none;
- **messageFromGcsPartnerToUav** = none;

```
1         uav : process UavControl(commanderId, poweredOff,
      ↪ varUav_connectionToGcsCommander,
      ↪ varGcs_isCommanderConnectionAuthorized,
      ↪ messageFromUavToGcsCommander,
      ↪ messageFromUavToGcsPartner,
      ↪ messageFromGcsCommanderToUav,
      ↪ messageFromGcsPartnerToUav,
      ↪ varUav_checkDistanceToGoal, commanderHasSwitched,
      ↪ keyboardPress);
2
```

Figure 5.2: NuSMV Initialization of Module uavControl in Module main

- **varGcs_isPartnerConnected** = FALSE;

State Transitions

The state transitions presented in Appendix A. All the state transitions are done per the requirements in section 3.6.

Properties

There are a total of 18 verified properties for the **Module uavControl**.
REQ-UAV-001 - The UAV shall proceed to the state "Start-up" when powered on.

```
1 CTLSPEC ((stateUAV = poweredOff & varUav_isPowerOn) -> EF(stateUAV
      ↪ = startUp))
```

This CTL property specifies that if the UAV is powered on while in the "Powered Off" state, it should eventually reach the "Start-up" state. The EF operator ensures that the "Start-up" state is reachable from the current state.
REQ-UAV-002 - If the UAV is in the "Start-up" state and is powered off, then it shall transition to the "Power Off" state.

```
1 LTLSPEC ((stateUAV = startUp) -> ((varUav_isPowerOn & !next(
      ↪ varUav_isPowerOn)) -> F(stateUAV = poweredOff)))
```

This LTL property ensures that if the UAV is in the "Start-up" state and the power is turned off, it will eventually transition to the "Powered Off" state. The F operator ensures that the "Powered Off" state will be reached at some point in the future.
REQ-UAV-003 - If the UAV is in the "Operating" state and is powered off, then it shall transition to the "Power Off" state.

```
1 LTLSPEC G((stateUAV = operational) -> ((varUav_isPowerOn & !next(
      ↪ varUav_isPowerOn)) -> X(stateUAV = poweredOff)))
```

This LTL property asserts that globally, whenever the UAV is in the "Operational" state and the power is turned off, the next state must be "Powered Off". The X operator enforces that this transition occurs immediately in the next state.
REQ-UAV-006 - When the UAV connects to the configured Ground Control Station, it shall transition to the "Operating" state.

```
1 LTLSPEC G((varUav_connectionToGcsCommander = connected) -> F(
      ↪ stateUAV = operational))
2 LTLSPEC G((varUav_isPowerOn & varUav_isConfigurationLoaded &
      ↪ varUav_connectionToGcsCommander = connected) -> F(
      ↪ stateUAV = operational))
```

The first LTL property ensures that whenever the UAV connects to the Ground Control Station (GCS), it will eventually transition to the "Operational" state. The second property adds more specificity by requiring that the UAV is powered on and the configuration is loaded before it transitions to the "Operational" state.

REQ-UAV-007 - When the UAV transitions to the "Operating" state, it shall connect to the flight controller. In addition, it shall keep retrying the connection request until it is successful.

```
1 LTLSPEC G((stateUAV = operational) -> F(stateUAV_telemetry =
      ↪ connecting))
```

This LTL property guarantees that whenever the UAV is in the "Operational" state, it will eventually attempt to connect to the flight controller.

REQ-UAV-008 - When the UAV is in the "Operating" state and connected to the flight controller, it shall get current telemetry and transmit it to the ground control station.

```
1 LTLSPEC G((stateUAV = operational) -> F(stateUAV_telemetry =
      ↪ telemetry))
```

This LTL property ensures that whenever the UAV is in the "Operational" state, it will eventually enter the "Telemetry" sub-state.

REQ-UAV-009 - While the UAV transitions to the "Operating" state, it shall listen continuously for incoming messages from ground control stations.

```
1 LTLSPEC G (stateUAV = startUp & next(stateUAV) = operational ->
      ↪ stateUAV_messageHandling = listening)
```

This LTL property asserts that during the transition from the "Start-up" to the "Operational" state, the UAV must be in a listening state for incoming messages from the ground control stations.

REQ-UAV-010 - While the UAV is in the "Operating" state, it shall validate any incoming messages from ground control stations.

```
1 CTLSPEC AG (stateUAV = operational & varUav_newReceivedMessage ->
      ↪ AX (stateUAV_messageHandling = validating))
```

This CTL property guarantees that for all paths, if the UAV is in the "Operational" state and there is a new incoming message, the UAV will immediately transition to the "Validating" sub-state.

REQ-UAV-011 - While the UAV is in the "Operating" state, it shall discard any invalid incoming messages from ground control stations.

```
1 LTLSPEC G (stateUAV = operational & varUav_newReceivedMessage & !
      ↪ varUav_isReceivedMessageValid ->
      ↪ stateUAV_messageHandling = listening)
```

This LTL property ensures that if the UAV receives an invalid message while in the "Operational" state, it will return to the listening state, implicitly indicating that the message has been discarded.
REQ-UAV-012 - While the UAV is in the "Operating" state, if the message type is "Keyboard Command", it shall perform a movement maneuver according to the content of the message.

```
1 CTLSPEC EX ((stateUAV = operational & varUav_isReceivedMessageValid
    ↪ & varUav_messageType = keyboardCommand) -> (
    ↪ varUav_performMovement = Up | varUav_performMovement =
    ↪ Down | varUav_performMovement = Left |
    ↪ varUav_performMovement = Right))
```

This CTL property ensures that if the UAV is in the "Operational" state and receives a valid keyboard command, it will perform the specified movement (Up, Down, Left, or Right) in the next state.
REQ-UAV-014 - While the UAV is in the "Operating" state, if the message type is "Mission", it shall set the distance to the goal.

```
1 CTLSPEC AG (stateUAV = operational & varUav_messageType = mission &
    ↪ varUav_isReceivedMessageValid -> AX (
    ↪ varUav_isGoalDistanceMeasurementEnabled))
2
3 CTLSPEC AG (stateUAV = operational & varUav_messageType = mission &
    ↪ varUav_isGoalDistanceMeasurementEnabled -> AX (
    ↪ varUav_checkDistanceToGoal >=0 &
    ↪ varUav_checkDistanceToGoal <=99 ))
```

The first CTL property ensures that if the UAV is operational and the mission message is valid, the goal distance measurement will be enabled in the next state. The second property checks that once the goal distance measurement is enabled, the calculated distance is within the range of 0 to 99 meters.
REQ-UAV-015 - While the UAV is in the "Operating" state, if the message type is "Handover - Connect to Partner as Client" and the UAV is within the handover region, it shall send a message request to connect to the partner. In addition, it shall retry to transmit the message in case of failure.

```
1 LTLSPEC G (stateUAV = operational & varUav_messageType =
    ↪ handoverConnectToPartnerAsClient &
    ↪ varUav_isInHandoverRegion -> (
    ↪ varUav_sendConnectRequestToPartner & X (
    ↪ varUav_sendConnectRequestToPartner |
    ↪ varUav_retryConnectRequest)))
```

This LTL property guarantees that if the UAV is in the "Operational" state, the message type is "Handover - Connect to Partner as Client", and the UAV is within the handover region, it will either send a connect request to the partner or retry sending the request in case of failure.
REQ-UAV-016 - While the UAV is in the "Operating" state, after successfully transmitting the message request to connect to the partner, it shall transmit a handover request to the partner. In addition, it shall retry transmitting the message in case of failure.

```
1 LTLSPEC G (stateUAV = operational &
      ↪ varGcs_acknowledgeConnectMessage -> (
      ↪ varUav_sendHandoverRequestToPartner & X (
      ↪ varUav_sendHandoverRequestToPartner |
      ↪ varUav_retryHandoverRequestToPartner)))
```

This LTL property ensures that if the UAV successfully transmits the connect request and receives acknowledgment, it will proceed to send the handover request and will retry if the transmission fails.
REQ-UAV-017 - While the UAV is in the "Operating" state, after successfully transmitting the connect handover request to the partner and disconnecting from the commander, it shall change the commander to the current partner.

```
1 LTLSPEC G (stateUAV = operational &
      ↪ varUav_sendHandoverRequestToPartner &
      ↪ varGcs_authorizationFromCommanderForHandover &
      ↪ varUav_currentGcsCommander = 1 -> next(
      ↪ varUav_currentGcsCommander) != 1)
```

This LTL property ensures that after successfully transmitting the handover request, and receiving authorization from the commander, the UAV will change its current Ground Control Station (GCS) commander from the previous one.
REQ-UAV-018 - During the handover procedure, the UAV shall remain operational until the handover procedure is complete.

```
1 LTLSPEC G((stateUAV_handoverProcedure = onGoing) -> (stateUAV =
      ↪ operational -> stateUAV_handoverProcedure != off))
```

This LTL property asserts that if the handover procedure is ongoing, the UAV must remain operational, and the handover procedure should not be turned off until completion.

5.1.2 Ground control station (GCS) module

The module is used to create a process in the Module main with variables created in the Module main through a process:

```
1    MODULE GcsControl (id, commanderId, staceGcs,
      ↪ varUav_connectionToGcsCommander,
      ↪ varGcs_isCommanderConnectionAuthorized,
      ↪ varGcs_messageTypeCommanderToReceive,
      ↪ varGcs_messageTypePartnerToReceive,
      ↪ varGcs_messageTypeCommanderToTransmit,
      ↪ varGcs_messageTypePartnerToTransmit,
      ↪ varUav_checkDistanceToGoal,varGcs_isPartnerConnected,
      ↪ commanderHasSwitched, keyboardPress)
2
3
```

Figure 5.3: Module GcsControl definition

Each variable defined for the **Module GcsControl** is used as an interface, which means the **Module main** can define variables that can be inserted in the module. This can be viewed

as the **Module GcsControl** is a subfunction of the **Module main**. The definition of each variable is the following:

- **id**: Defines the ID of the GCS. Scale/Interpretation: 1 = GCS A, 2 = GCS B;
- **commanderId**: Defines the ID of the GCS Commander. Scale/Interpretation: 1 = GCS A, 2 = GCS B;
- **stateGcs**: poweredOff, startUp, operational; – Possible states for the GCS system. Scale/Interpretation: poweredOff = GCS is powered off, startUp = GCS is in start-up procedure, operational = UAV is in operational state;
- **varUav_connectionToGcsCommander**: Defines that the UAV is connected to a GCS (commander). Scale/interpretation: noConnection = No connection available; attemptingToConnect = Has requested connection to the GCS, connected = Is currently connected to the GCS commander;
- **varGcs_isCommanderConnectionAuthorized**: Defines the authorization for a start-up connection for the UAV to connect to the GCS commander. Scale/Interpretation: TRUE = Authorized connection, FALSE = Not authorized;
- **varUav_messageTypeToTransmitToCommander**: none, telemetry, mission, onHandover; – Defines the possible message types to send from UAV to the GCS. Scale/Interpretation: none = No message available, telemetry = Telemetry type message, mission = Mission type message, onHandover = Proceeding with handover procedure;
- **varUav_messageTypeToTransmitToPartner**: Defines the possible message types to send from UAV to the GCS. Scale/Interpretation: none = No message available, handoverRequest = Request handover to GCS Partner, connectHandover = Connect to GCS Partner;
- **varUav_messageTypeToReceiveFromCommander**: Defines the possible message types to send from UAV to the GCS. Scale/Intepretation: none = No message available, telemetry = Telemetry request message, mission = Mission message type, okHandover = Authorize handover to connected partner, keyboardCommand = Keyboard type command to perform movement;
- **varUav_messageTypeToReceiveFromPartner**: Defines the possible message types to send from UAV to the GCS. Scale/Interpretation: none = No message available, acknowledgeConnectMessage = Accept connect message;
- **varUav_checkDistanceToGoal**: Defines the distance to goal as an integer. Scale/interpretation: 1 = 1 meter; Range = 0..99 meters;
- **commanderHasSwitched**: Defines if the commander has switched. Scale/Interpretation: TRUE = Commander has switched during the current handover procedure, FALSE = Commander has not switched during the current handover procedure;
- **keyboardPress**: Possible directions for the keyboard presses. Scale/Interpretation: none = No key is being pressed, down = Downwards movement, up = Upwards movement, left = Left movement, right = Right movement;

Initialization

The module is used to create a process in the **Module main** with variables created in the **Module main**.
For the initialization, the initial value for the variables are the following:

- **commanderId** = 1;
- **stateGcs** = poweredOff;
- **varUav_connectionToGcsCommander** = noConnection;

```
1     gcs1 : process GcsControl(1, commanderId, poweredOff,
        ↪ varUav_connectionToGcsCommander,
        ↪ varGcs_isCommanderConnectionAuthorized,
        ↪ messageFromUavToGcsCommander,
        ↪ messageFromUavToGcsPartner,
        ↪ messageFromGcsCommanderToUav,
        ↪ messageFromGcsPartnerToUav,
2     varUav_checkDistanceToGoal,varGcs_isPartnerConnected,
        ↪ commanderHasSwitched, keyboardPress);
3
4     gcs2 : process GcsControl(2,  commanderId, poweredOff,
        ↪ varUav_connectionToGcsCommander,
        ↪ varGcs_isCommanderConnectionAuthorized,
        ↪ messageFromUavToGcsCommander,
        ↪ messageFromUavToGcsPartner,
        ↪ messageFromGcsCommanderToUav,
        ↪ messageFromGcsPartnerToUav,
5     varUav_checkDistanceToGoal,varGcs_isPartnerConnected,
        ↪ commanderHasSwitched, keyboardPress);
6
```

Figure 5.4: NuSMV Initialization of Module GcsControl in Module main

- **varGcs_isCommanderConnectionAuthorized** = FALSE;
- **messageFromUavToGcsCommander** = none;
- **messageFromUavToGcsPartner** = none;
- **messageFromGcsCommanderToUav** = none;
- **messageFromGcsPartnerToUav** = none;
- **varGcs_isPartnerConnected** = FALSE;

State Transitions

The state transitions presented in Appendix A. All the state transitions are done per the requirements in section 3.6.

Properties

There are a total of 18 verified properties for the Module GcsControl.
REQ-GCS-001 - The GCS shall proceed to state "Start-up" when powered on.

```
1 CTLSPEC ((stateGcs = poweredOff & varGcs_isPowerOn) -> EF(stateGcs
        ↪ = startUp))
```

This CTL property ensures that if the GCS is in the "Powered Off" state and the power is turned on, then the GCS will eventually reach the "Start-up" state. The EF operator in CTL is used to express that there exists a future path where the "Start-up" state will be reached.
REQ-GCS-002 - If the GCS is in the "Start-up" state and is powered off, then it shall transition to "Power Off" state.

```
1 LTLSPEC ((stateGcs = startUp) -> ((varGcs_isPowerOn & !next(
          ↪ varGcs_isPowerOn)) -> F(stateGcs = poweredOff)))
```

This LTL property specifies that if the GCS is in the "Start-up" state and the power is turned off (!next(varGcs_isPowerOn)), the system will eventually reach the "Powered Off" state. The F (eventually) operator in LTL ensures that the "Powered Off" state will be reached in some future state.

REQ-GCS-003 - If the GCS is in the "Operating" state and is powered off, then it shall transition to "Power Off" state.

```
1 LTLSPEC ((stateGcs = operational) & !varGcs_isPowerOn -> next(
          ↪ stateGcs = poweredOff) U varGcs_isPowerOn)
```

This LTL property asserts that if the GCS is in the "Operating" state and the power is turned off, the system will transition to the "Powered Off" state, and it will remain in that state until power is restored (U varGcs_isPowerOn). The U (until) operator is used here to specify that the system will be in the "Powered Off" state until the power is turned back on.

REQ-GCS-004 - When the GCS transitions to "Start-up" state, it shall load up the parameters from the configuration data.

```
1 LTLSPEC G((varGcs_isPowerOn & varGcs_isConfigurationLoaded &
          ↪ varUav_connectionToGcsCommander = connected) -> next(
          ↪ stateGcs = operational) U (varGcs_isPowerOn &
          ↪ varGcs_isConfigurationLoaded &
          ↪ varUav_connectionToGcsCommander = connected))
```

This LTL property ensures that once the GCS has power, the configuration is loaded, and the UAV is connected, it will eventually transition to the "Operational" state. The G (globally) operator asserts that this condition will hold true throughout the execution of the system.

REQ-GCS-005 - When the GCS is in "Start-up" state and has loaded configuration data, it shall connect to the configured UAV.

```
1 LTLSPEC (stateGcs = startUp & varGcs_isConfigurationLoaded &
          ↪ varUav_connectionToGcsCommander = attemptingToConnect
          ↪ & varGcs_isCommanderConnectionAuthorized & id =
          ↪ commanderId -> X(varUav_connectionToGcsCommander =
          ↪ connected))
```

This LTL property specifies that if the GCS is in the "Start-up" state with the configuration loaded and the UAV is attempting to connect, the GCS will connect to the UAV in the next state. The X (next) operator is used to assert that this connection happens in the immediate next state.

REQ-GCS-006 - When the GCS has connected to the configured UAV, it shall transition to "Operating" state.

```
1 LTLSPEC (stateGcs = startUp & varGcs_isConfigurationLoaded &
          ↪ varUav_connectionToGcsCommander = attemptingToConnect
          ↪ & varGcs_isCommanderConnectionAuthorized & id =
          ↪ commanderId -> X(stateGcs = operational &
          ↪ varUav_connectionToGcsCommander = connected))
```

This LTL property ensures that once the GCS has connected to the UAV, the system will transition to the "Operational" state in the next step. The X operator indicates that the transition occurs immediately after the conditions are met.

REQ-GCS-007 - When the GCS transitions to "Operating" state, it shall continuously listen to incoming messages.

```
1 LTLSPEC (stateGcs = operational -> X(stateGcs_messageHandling =
    ↪ listening))
```

This LTL property ensures that once the GCS enters the "Operating" state, it will begin listening for incoming messages in the next state. The X operator specifies that this listening behavior will occur immediately after entering the "Operating" state.

REQ-GCS-008 - While the GCS is in "Operating" state, it shall validate any incoming messages from the UAV.

```
1 LTLSPEC G(stateGcs = operational & varGcs_newReceivedMessage &
    ↪ stateGcs_messageHandling = listening -> (
    ↪ stateGcs_messageHandling = validating))
```

This LTL property asserts that whenever the GCS is in the "Operating" state and receives a new message, the message will always be validated. The G (globally) operator indicates that this validation will consistently happen as long as the system is in the "Operating" state.

REQ-GCS-009 - While the GCS is in "Operating" state, it shall discard any invalid incoming messages from the UAV.

```
1 LTLSPEC G(stateGcs = operational & varGcs_newReceivedMessage &
    ↪ stateGcs_messageHandling = validating & !
    ↪ varGcs_isNewMessageValid -> !(stateGcs_messageHandling
    ↪  = executing) U (stateGcs_messageHandling = validating
    ↪  & varGcs_isNewMessageValid))
```

This LTL property ensures that if the GCS is in the "Operating" state and receives an invalid message, it will not execute that message until a valid message is received. The U (until) operator is used to specify that the system will not transition to an execution state until the message is validated as correct.

REQ-GCS-010 - While the GCS is in "Operating" state and received message type is "Telemetry", it shall display the latest telemetry data.

```
1 LTLSPEC G(stateGcs = operational & stateGcs_messageHandling =
    ↪ executing & id = commanderId &
    ↪ varGcs_messageTypeCommanderToReceive = telemetry ->
    ↪ varGcs_displayLatestTelemetry)
```

This LTL property asserts that if the GCS is in the "Operating" state and a telemetry message is received, it will always display the latest telemetry data. The G (globally) operator ensures that this display action will occur whenever these conditions are met.

REQ-GCS-011 - While the GCS is in "Operating" state and "check route" is configured, it shall analyze the latest telemetry to verify if the UAV is currently on route.

```
1 LTLSPEC G(stateGcs = operational & varGcs_isCheckRouteConfigured &
    ↪ varGcs_displayLatestTelemetry -> varGcs_isUavOnRoute)
```

This LTL property ensures that if the GCS is in the "Operating" state, has the "check route" feature configured, and displays the latest telemetry, it will verify if the UAV is on the correct route. The G operator ensures that this check happens consistently whenever the conditions are true.

REQ-GCS-012 - While the GCS is in "Operating" state and received message type is "Connect", it shall set up the UAV connection and transmit an acknowledgment message to the UAV upon a successful connection setup.

```
1 LTLSPEC (stateGcs = startUp & varGcs_isConfigurationLoaded &
          ↪ varUav_connectionToGcsCommander = attemptingToConnect
          ↪  & varGcs_isCommanderConnectionAuthorized & id =
          ↪ commanderId -> X(stateGcs = operational &
          ↪ varUav_connectionToGcsCommander = connected))
```

This LTL property ensures that if the GCS is in the "Start-up" state and the UAV is attempting to connect, the system will transition to "Operational" and establish a connection to the UAV in the next state. The X operator specifies that this transition happens immediately after the conditions are met.

REQ-GCS-013 - While the GCS is in "Operating" state and received message type is "Mission", it shall load the distance to the mission goal from UAV.

```
1 LTLSPEC (stateGcs = operational &
          ↪ varGcs_messageTypeCommanderToReceive = mission &
          ↪ varUav_checkDistanceToGoal = 10 ->
          ↪ varGcs_missionDistance = 10)
```

This LTL property specifies that if the GCS is in the "Operating" state and receives a mission message, it will load the distance to the mission goal as provided by the UAV. The property asserts that this will happen whenever the mission message is received.

REQ-GCS-014 - While the GCS is in "Operating" state and received message type is "Handover Request", it shall transmit an acknowledgment message to the UAV.

```
1 LTLSPEC (stateGcs = operational &
          ↪ varGcs_messageTypePartnerToReceive = handoverRequest &
          ↪  !(id = commanderId) & varGcs_isNewMessageValid ->
          ↪ varGcs_messageTypePartnerToTransmit =
          ↪ acknowledgeConnectMessage)
```

This LTL property ensures that if the GCS receives a handover request while in the "Operating" state, it will transmit an acknowledgment message to the UAV. This property ensures that the GCS acknowledges the handover request as soon as it receives a valid message.

REQ-GCS-015 - While the GCS is in "Operating" state and received message type is "On Handover", it shall transmit an "OK Handover" to the UAV.

```
1 LTLSPEC (stateGcs = operational &
          ↪ varGcs_messageTypeCommanderToReceive = onHandover
2 & !(id = commanderId) & varGcs_isNewMessageValid  ->
          ↪ varGcs_messageTypeCommanderToTransmit = okHandover)
```

This LTL property ensures that when the GCS is in the "Operating" state and receives an "On Handover" message, it will respond with an "OK Handover" message, provided the message is valid.

REQ-GCS-016 - While the GCS is in "Operating" state and received message type is "Connect Handover", it shall set UAV communication as client and transmit a "UAV Connected" to the UAV.

```
1 LTLSPEC (stateGcs = operational &
          ↪ varGcs_messageTypePartnerToReceive = connectHandover
          ↪ -> varGcs_isPartnerConnected)
```

This LTL property specifies that when the GCS is in the "Operating" state and receives a "Connect Handover" message, it will establish communication with the UAV as a client and indicate that the UAV is connected.
REQ-GCS-018 - When the GCS transitions to "Operating" state and it detects a keyboard command, it shall transmit a validated command to the UAV.

```
1 LTLSPEC (stateGcs = operational & !(keyboardPress = none) ->
          ↪ varGcs_messageTypeCommanderToTransmit =
          ↪ keyboardCommand)
```

This LTL property ensures that if the GCS is in the "Operating" state and detects a keyboard command, it will transmit a validated command to the UAV. The property makes sure that any keyboard input is translated into a corresponding command sent to the UAV.

5.1.3 Module main

The module is used to create the process related to the UAV and the GCS. It is also used as a supervisor for all modules.

- **commanderId:** Definition of the commander Id. Scale/Interpretation: The number represents the ID of the GCS in control.
- **varUav_connectionToGcsCommander:** Defines that the UAV is connected to a GCS (commander). Scale/interpretation: noConnection = No connection available; attemptingToConnect = Has requested connection to the GCS, connected = Is currently connected to the GCS commander.
- **varGcs_isPartnerConnected:** Defines if the partner GCS is connected to the UAV at the end of the handover procedure.
- **varGcs_isCommanderConnectionAuthorized:** Defines the authroization for a start-up connection for the UAV to connect to the GCS commander. Scale/Interpretation: TRUE = Authorized connection, FALSE = Not authorized.
- messageFromUavToGcsCommander: none, telemetry, mission, onHandover; – Defines the possible message types to send from UAV to the GCS.
- **messageFromUavToGcsPartner:** none, handoverRequest, connectHandover; – Defines the possible message types to send from UAV to the GCS.
- **messageFromGcsCommanderToUav:** none, telemetry, mission, okHandover, keyboardCommand; – Defines the possible message types to send from UAV to the GCS.
- **messageFromGcsPartnerToUav:** none, acknowledgeConnectMessage; Defines the possible message types to send from UAV to the GCS.

Initialization

- **commanderId:** 1;
- **varUav_connectionToGcsCommander:** connected;
- **varGcs_isPartnerConnected:** boolean; – Defines if the partner GCS is connected to the UAV at the end of the handover procedure;

- **varGcs_isCommanderConnectionAuthorized:** Defines the authroization for a start-up connection for the UAV to connect to the GCS commander. Scale/Interpretation: TRUE = Authorized connection, FALSE = Not authorized;
- **messageFromUavToGcsCommander:** none;
- **messageFromUavToGcsPartner:** none;
- **messageFromGcsCommanderToUav:** none;
- **messageFromGcsPartnerToUav:** none;

State Transitions

The state transitions presented in Appendix A. All the state transitions are done per the requirements in section 3.6.

Properties

There are a total of 2 properties verified in the Module main.
REQ-GCS-017 - While the GCS Commander is in "Operating" state and GCS Partner is connected to UAV, the commander shall disconnect and the partner shall be the new commander.

```
1    LTLSPEC G(varUav_connectionToGcsCommander = connected &
         ↪ varGcs_isPartnerConnected & commanderId  = 1 & !
         ↪ commanderHasSwitched -> next(commanderId = 2))
2    LTLSPEC G(varUav_connectionToGcsCommander = connected &
         ↪ varGcs_isPartnerConnected & commanderId  = 1 & !
         ↪ commanderHasSwitched -> next(commanderId = 1))
```

Both LTL properties asserts if the UAV is connected to the GCS commander and the GCS partner is connected to the UAV, then it will eventually switch commanders.

5.1.4 Results

All requirements have at least 1 property verified. It is difficult to ascertain properties due to the complexity of the model which is caused by a significant state explosion[1]. There are certain issues such as the UAV being able to power off, which at any given moment affect most of the defined properties, large amount of states/variables, which affect the comprehension of the model and how difficult is to define actions that are not defined, such as a timeout, as it can only be added as a variable to define that it was not possible to transmit/receive a message due to a timeout that could occur for various of reasons (interference, software delay, lost packets). There are two main concerns with defined protocol at the moment:

- There is no way to cancel the handover procedure once started until it finishes.
- There is no constant verification if the UAV is in the handover region.

The first issue can be solved by adding a timeout/cancel operation that jumps to a state similar to idle. The cancel operation may be added but it introduces unwanted complexity. Through abstraction, a variable timeout can be inserted and used to proceed for each state during the handover procedure. The second issue can follow a similar solution, where the variable isUavInHandoverRegion is verified throughout the states, therefore, if the UAV ever leaves the region, the handover procedure can be cancelled. Taking this into account, it is necessary to define how to proceed with the second iteration.

[1] https://ieeexplore.ieee.org/document/5369392

The second iteration must be less complex and focus on the communication protocol itself and not concern about hardware interactions.

5.2 Second Iteration of the model

After reviewing the previous iteration, it was necessary to understand where to focus the analysis. Since the focus is in the analysis of the authority transfer protocol, it is possible to abstract from hardware and focus solely on a functional level. The model is represented in Appendix B.

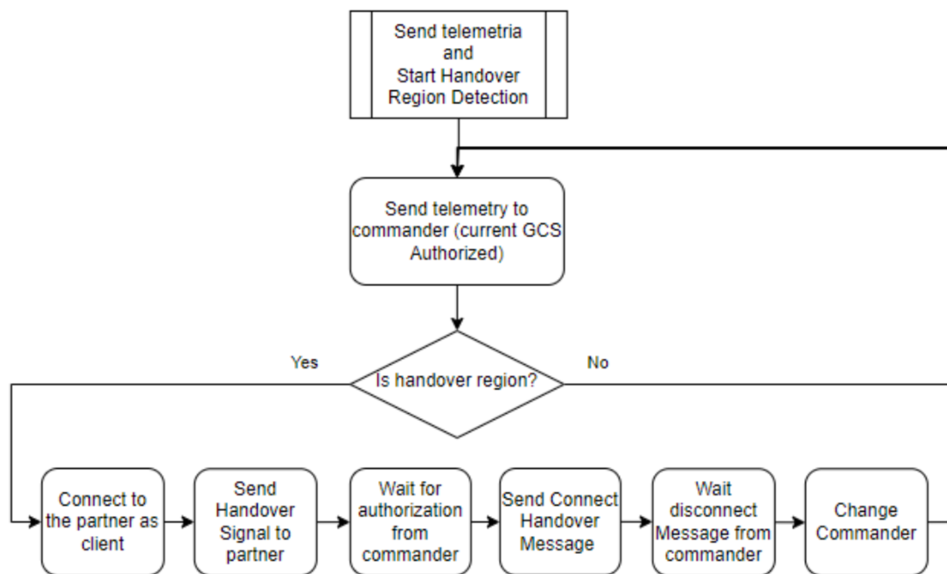


Figure 5.5: Handover procedure available in (Ferreira 2023)

It is possible to specify the model only using the two diagrams in figure 5.5 and 5.6. For figure 5.6, only the Request Handover to Disconnect operation (inclusive) is necessary to approach the authority transfer protocol.
Focusing only on three entities: the UAV, the GCS Commander and the GCS Partner, requirements can be derived for each one.

5.2.1 Derived Requirements

Unmanned Aerial Vehicle

- **REQ-UAV-001:** The UAV shall request handover to the GCS Partner, when it reaches the handover region.
- **REQ-UAV-002:** The UAV shall connect to the GCS Partner, if the GCS Commander authorizes the handover to the GCS Partner.

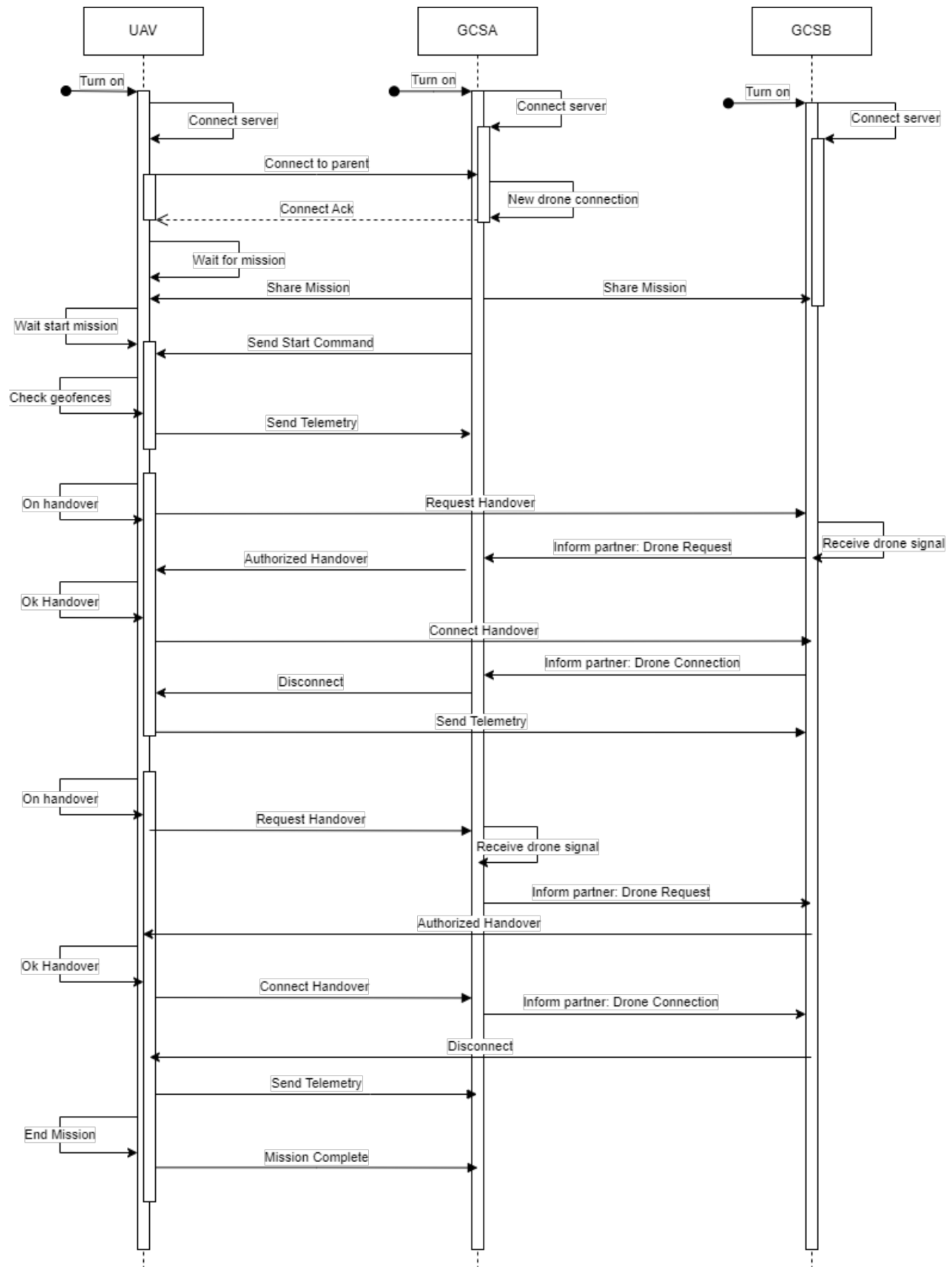


Figure 5.6: Handover procedure action diagram available in (Ferreira 2023)

Ground Control Station

- **REQ-GCS-001:** The GCS Commander shall authorize handover, if the GCS Partner informed the GCS Commander that the UAV has requested handover to the GCS Partner.
- **REQ-GCS-002:** The GCS Commander shall disconnect from the UAV, transfering the authority over the UAV to the GCS Partner, when the GCS Partner informs the GCS Commander that the UAV has connected to the GCS Partner.
- **REQ-GCS-003:** The GCS Partner shall inform GCS Commander that the UAV has requested handover when the GCS Partner receives a handover request from the UAV.
- **REQ-GCS-004:** The GCS Partner shall inform GCS Commander that the UAV has connected to the GCS Partner, when the UAV connects to the GCS Partner.

5.2.2 NuSMV model

The NuSMV model for iteration 2 was created from scratch, discarding the previous model since it would require more effort to update it than to create from an empty file.
The NuSMV model is composed by 3 modules: *uav*, *gcs* and *main*. All modules are ran with FAIRNESS.

Module main

The modules *uav* and *gcs* are called as processes in the module *main*.

```
1    -- Define Processes
2    uav : process uav(stateUav, stateGcs, isUavInHandoverRegion);
3    gcs1 : process gcs(1,commanderId, stateUav, stateGcs,
         ↪ isUavInHandoverRegion,
4    commanderHasSwitched);
5    gcs2 : process gcs(2,commanderId, stateUav, stateGcs,
         ↪ isUavInHandoverRegion,
6    commanderHasSwitched);
```

The module has a total of 6 variables:

- **commanderId:** Identifies the ID of the commander. Scale/Interpretation:
  - 1 = GCS A;
  - 2 = GCS B;
- **isUavInHandoverRegion:** Defines if the UAV is in the handover region. Scale Interpretation:
  - TRUE = UAV is in handover region;
  - FALSE = UAV is not in handover region;
- **stateUav:** Defines the state of the UAV. Scale/Interpretation:
  - idle = Idle mode, awaiting instructions;
  - receiveTelemetry = Receive telemetry information from UAV;
  - requestHandover = UAV is requesting handover to GCS Partner;
  - connectHandover = UAV connects to GCS Partner after it has been authorized for handover by GCS Commander;
- **stateGcs:** Defines the state of the GCS (Commander and Partner). Scale/Interpretation:
  - idle = Idle mode, awaiting instructions;
  - receiveTelemetry = Transmit telemetry information to GCS Commander;

- acceptPartnerRequest = GCS Partner has received a handover request from UAV;
- authorizeHandover = GCS Commander has authorized, after receiving information regarding a connection request from UAV to GCS Partner from GCS Partner;
- droneConnectToPartner = GCS Partner connects to UAV after the UAV has been authorized to connect to the GCS;
- disconnectCommander= GCS Commander disconnects from UAV after the UAV has connected to the GCS Partner and transfers authority to the GCS Partner;

- **commanderHasSwitched** - Auxiliary variable to inform the procedure has reached the final step. Scale/Intepretation:
  - TRUE = Commander has switched for the current handover procedure;
  - FALSE = Commander has not switched for current the handover procedure;
- **messageTimeout** - Defines if there has been a message timeout between two entities (Any combination of UAV/GCS Commander/GCS Partner);

Initialization of the module:

```
1     init(commanderId) := 1;
2     init(isUavInHandoverRegion) := TRUE;
3     init(stateUav) := idle;
4     init(stateGcs) := idle;
5     init(commanderHasSwitched) := FALSE;
6     init(messageTimeout) := FALSE;
```

Only the variables *isUavInHandoverRegion*, *commanderId* and *commanderHasSwitched* transition in the module *main* and the states of UAV and GCSs are updated in the respective modules.

```
1     next(isUavInHandoverRegion) :=
2         case
3             TRUE: {TRUE,FALSE};
4             --TRUE: TRUE;
5         esac;
```

The *isUavInHandoverRegion* transition can go TRUE or FALSE at every state. The reason for this is to evaluate how the model interacts if the UAV can move to or move away from the handover region.

```
1     next(commanderId) :=
2         case
3             commanderId = 1 & stateGcs = disconnectCommander & !
           ↪ commanderHasSwitched : 2;
4             commanderId = 2 & stateGcs = disconnectCommander & !
           ↪ commanderHasSwitched : 1;
5             TRUE: commanderId;
6         esac;
```

Whenever the commanderId is 1 and the GCS commander is in disconnectCommander state and the commanderHasSwitched is false, then it shall switch from 1 to 2. This can also happen in the inverse way.

```
1     next(commanderHasSwitched) :=
2         case
3             stateGcs = disconnectCommander : TRUE;
4             TRUE: FALSE;
5         esac;
```

Whenever the GCS Commander reaches the state disconnectCommander, it shall be TRUE. This is to guarantee that the switch only happens once per handover.

Module uav

The module uav does not have any assigned variables as it uses the variables defined in the module main.
It only possesses a single state transition (stateUav):

```
1     next(stateUav) := case
2         stateUav = requestHandover & stateGcs = authorizeHandover :
            ↪  connectHandover;
3         stateUav = sendTelemetry & isUavInHandoverRegion :
            ↪ requestHandover;
4         stateUav = idle | stateGcs = receiveTelemetry:
            ↪ sendTelemetry;
5         stateUav = sendTelemetry | stateGcs = disconnectCommander :
            ↪  idle;
6         TRUE : stateUav;
7     esac;
```

The UAV shall initiate in state idle and transition only to sendTelemetry if the GCS Commander is expecting to receive the telemetry. This is an abstract way to model as in reality the UAV would constantly be sending the telemetry to the GCS Commander.
It shall transition to requestHandover, if the UAV is in sendTelemetry (to simulate latest telemetry result) and is in the handover region.
It shall await authorization of the handover from GCS Commander, and when the authorization is received, then transition to state connectHandover.
Finally, when the GCS Commander disconnects, it shall go to idle state, finishing the cycle.

Module gcs

The module gcs does not have any assigned variables as it uses the variables defined in the module main.
It only possesses a single state transition (stateGcs):

```
1      next(stateGcs) :=
2       case
3          stateGcs = droneConnectToPartner & (commanderId = id) & !
             ↪ commanderHasSwitched : disconnectCommander;
4          stateGcs = authorizeHandover & stateUav = connectHandover &
             ↪  !(commanderId = id) : droneConnectToPartner;
5          stateGcs = acceptPartnerRequest & (commanderId = id) :
             ↪ authorizeHandover;
6          stateGcs = receiveTelemetry & isUavInHandoverRegion & !(
             ↪ commanderId = id) & stateUav = requestHandover:
             ↪ acceptPartnerRequest;
7          stateGcs = idle & commanderId = id : receiveTelemetry;
8          stateGcs = receiveTelemetry | (stateGcs =
             ↪ disconnectCommander & commanderHasSwitched) : idle;
9          TRUE : stateGcs;
10      esac;
```

The GCS shall initiate in state idle and transition only to sendTelemetry if the GCS is the GCS Commander.
If the GCS is in state receiveTelemetry, the UAV is in handover region, the UAV is requesting handover and the GCS is not the commander (Partner), then it shall accept the partner request and go to state acceptPartnerRequest.
If the GCS is in state acceptPartnerRequest and it is the GCS Commander, then it shall transition to authorizeHandover. If the GCS is in state authorizeHandover, the UAV has requested to connect to the GCS Partner and the GCS is not the Commander (Partner), then it shall connect to the UAV as partner and transition to the state droneConnectToPartner.
If the GCS is in state droneConnectToPartner, the GCS is GCS Commander and the Commander switch hasn't happened, then the GCS Commander shall disconnect from the UAV.

Model Properties

```
1 LTLSPEC G (stateUav = requestHandover & stateGcs =
             ↪ acceptPartnerRequest -> F stateGcs = authorizeHandover
             ↪ );
```

This LTL specification ensures that when the UAV requests a handover and the GCS accepts the partner request, the system will eventually authorize the handover. The formula guarantees that this transition occurs whenever the conditions are met, ensuring proper handover procedure.

```
1 LTLSPEC (!isUavInHandoverRegion & commanderId = 1 & stateUav =
             ↪ requestHandover -> !isUavInHandoverRegion U (F ((
             ↪ commanderId = 2) & !isUavInHandoverRegion &
             ↪ commanderHasSwitched & stateGcs = disconnectCommander)
             ↪ ));
```

This LTL specification verifies that a commander switch is possible outside of the handover region. It ensures that when the UAV is outside the handover region and requests a handover, the system will not switch the commander until the conditions are met and the commander is successfully switched.

```
1 LTLSPEC (isUavInHandoverRegion & commanderId = 1 & stateUav =
        ↪ requestHandover & stateGcs = receiveTelemetry -> F (
        ↪ commanderId = 2));
```

This LTL specification ensures that when the UAV is within the handover region and starts the handover process, the GCS will eventually switch commanders from commanderId = 1 to commanderId = 2. This guarantees that the handover process is completed successfully when the UAV is in the correct region.

```
1 CTLSPEC EG (isUavInHandoverRegion & stateUav = idle -> EF (stateUav
        ↪  = requestHandover));
```

This CTL specification checks that there is always a path where the UAV, when in the handover region and idle state, will eventually start the handover process. It ensures that the system has the potential to initiate the handover when the UAV is in the appropriate region.

```
1 LTLSPEC G (stateUav = idle -> F stateUav = sendTelemetry);
```

This LTL specification ensures that whenever the UAV is in the idle state, it will eventually send telemetry data. This guarantees that telemetry data will be sent periodically while the UAV is idle.

```
1 LTLSPEC (isUavInHandoverRegion & stateUav = requestHandover -> F
        ↪ stateUav = connectHandover);
```

This LTL specification ensures that when the UAV is in the handover region and requests a handover, it will eventually connect for the handover. This guarantees that the UAV will follow through with the connection process during handover.

```
1 CTLSPEC (!isUavInHandoverRegion & stateUav = idle & stateGcs = idle
        ↪  -> EF commanderHasSwitched);
```

This CTL specification checks that when the UAV and GCS are both in the idle state and the UAV is not in the handover region, there exists a path where the commander switch will eventually occur. It ensures that the commander can be switched under these conditions.

```
1 LTLSPEC G (stateGcs = disconnectCommander & commanderId = 1 & !
        ↪ commanderHasSwitched -> F commanderId = 2);
```

This LTL specification ensures that when the GCS disconnects Commander 1 and the switch has not yet occurred, the system will eventually switch to Commander 2. This guarantees the completion of the commander switch.

```
1 CTLSPEC EG (stateGcs = idle & commanderId = 1 & !
        ↪ commanderHasSwitched -> EF commanderId = 2);
```

This CTL specification checks that when the GCS is in the idle state and commanderId = 1 without a switch having occurred, there exists a path where the system will eventually switch to Commander 2. It ensures that the switch is possible.

```
1 LTLSPEC G (stateGcs = disconnectCommander & commanderId = 2 & !
        ↪ commanderHasSwitched -> F commanderId = 1);
```

This LTL specification ensures that when the GCS disconnects Commander 2 and the switch has not yet occurred, the system will eventually switch back to Commander 1. This guarantees a reversible commander switch process.

```
1 LTLSPEC G (stateGcs = disconnectCommander & commanderId = 1 & !
          ↪ commanderHasSwitched -> F commanderId = 2 -> F(
          ↪ stateGcs = disconnectCommander & commanderId = 2 & !
          ↪ commanderHasSwitched) -> F(commanderId = 1));
```

This LTL specification ensures that after disconnecting Commander 1, the system will switch to Commander 2 and then back to Commander 1, following a specific sequence of operations. It verifies the ability to switch commanders in a controlled manner.

```
1 CTLSPEC EG (stateGcs = idle & commanderId = 2 & !
          ↪ commanderHasSwitched -> EF commanderId = 1 -> EF
          ↪ commanderId = 2);
```

This CTL specification checks that when the GCS is in the idle state with commanderId = 2 and no switch has occurred, there exists a path where the system can switch to Commander 1 and then back to Commander 2. It ensures that the system supports bidirectional commander switching.

```
1 CTLSPEC EG (stateUav = idle -> EF (stateUav = sendTelemetry));
```

This CTL specification ensures that the UAV can transition from the idle state to the state where it sends telemetry. It checks the system's ability to reach the telemetry state from idle.

```
1 CTLSPEC EG (stateUav = idle & isUavInHandoverRegion -> EF(stateUav
          ↪ = idle));
```

This CTL specification ensures that the UAV can return to the idle state from any state while in the handover region. It verifies that the idle state is reachable in the handover region.

```
1 CTLSPEC EG (stateUav = idle -> EF (stateUav = requestHandover));
```

This CTL specification ensures that the UAV can transition from the idle state to the request handover state. It checks that the handover request process can be initiated from idle.

```
1 CTLSPEC EG (stateUav = idle -> EF (stateUav = idle));
```

This CTL specification checks that the UAV can remain in or return to the idle state from any other state. It verifies that the idle state is always accessible.

```
1 CTLSPEC EG (stateUav = idle -> EF(stateUav = sendTelemetry));
```

This CTL specification ensures that the UAV can send telemetry data from the idle state. It checks that telemetry transmission is reachable from idle.

```
1 CTLSPEC EG (stateUav = idle -> EF (stateUav = connectHandover) ->
          ↪ EF (stateUav = idle));
```

This CTL specification ensures that the UAV can initiate a handover connection from the idle state and then return to idle. It verifies the process of connecting for handover and returning to idle.

```
1 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = receiveTelemetry));
```

This CTL specification checks that the GCS can transition from the idle state to the state where it receives telemetry. It verifies the GCS's ability to receive telemetry from idle.

```
1 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = acceptPartnerRequest))
      ↪ ;
```

This CTL specification ensures that the GCS can transition from the idle state to the state where it accepts a partner request. It verifies the ability to handle partner requests from idle.

```
1 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = authorizeHandover));
```

This CTL specification checks that the GCS can authorize a handover from the idle state. It verifies the ability to proceed with handover authorization from idle.

```
1 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = droneConnectToPartner)
      ↪ );
```

This CTL specification ensures that the GCS can connect the UAV to a partner system from the idle state. It verifies that the connection process is reachable from idle.

```
1 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = disconnectCommander)
      ↪ -> EF (stateGcs = idle));
```

This CTL specification checks that the GCS can disconnect a commander and return to the idle state. It verifies the process of disconnecting the commander and returning to idle.

5.2.3 Results

In this iteration, all requirements have at least one property successfully verified. The model is notably simpler compared to the previous version, with a greater emphasis on functionality, albeit in a more abstract manner.
A modification in this iteration was the introduction of a variable to account for message timeouts, which were not initially considered in the derived requirements. This variable was added without being incorporated into any state transitions, which ensured that it did not impact the properties already verified. However, it was observed that the verification process revealed that implementing checks for the absence of message transmission remains challenging, as this scenario is not explicitly addressed in the requirements.
Moreover, the verified properties indicate that it is possible to initiate the procedure within the handover region, exit the region, and still complete the procedure. This scenario needs to be addressed in the next iteration, and the requirements should be updated accordingly to reflect this case.
Aside from these two concerns, all other properties were proven to be consistent with the established diagrams and the derived requirements for this iteration. The results demonstrate that the model effectively captures the intended functionality while remaining aligned with the specified requirements for iteration two.
The next iteration must incorporate the solution for message timeout and constant verification of the handover region.

5.3 Third Iteration of the model

Focusing only on three entities: the UAV, the GCS Commander and the GCS Partner, requirements can be derived for each one. The added requirements are in bold in relation to the derived requirements in interation 2. The model is represented in Appendix C.

5.3.1 Derived Requirements

Unmanned Aerial Vehicle

- **REQ-UAV-001:** The UAV shall request handover to the GCS Partner, when it reaches the handover region.
- **REQ-UAV-002:** The UAV shall connect to the GCS Partner, if the GCS Commander authorizes the handover to the GCS Partner.
- **REQ-UAV-003: The UAV shall transition to state idle if a message timeout occurred or if the UAV is not in handover region.**

Ground Control Station

- **REQ-GCS-001:** The GCS Commander shall authorize handover, if the GCS Partner informed the GCS Commander that the UAV has requested handover to the GCS Partner.
- **REQ-GCS-002:** The GCS Commander shall disconnect from the UAV, transfering the authority over the UAV to the GCS Partner, when the GCS Partner informs the GCS Commander that the UAV has connected to the GCS Partner.
- **REQ-GCS-003:** The GCS Partner shall inform GCS Commander that the UAV has requested handover when the GCS Partner receives a handover request from the UAV.
- **REQ-GCS-004:** The GCS Partner shall inform GCS Commander that the UAV has connected to the GCS Partner, when the UAV connects to the GCS Partner.
- **REQ-GCS-005: The GCS shall transition to state idle if a message timeout occurred or if the UAV is not in handover region.**

5.3.2 NuSMV model

The NuSMV model for iteration 2 was created from scratch, discarding the previous model since it would require more effort to update it than to create from an empty file.
The NuSMV model is composed by 3 modules: *uav*, *gcs* and *main*. All modules are ran with FAIRNESS.

Module main

The modules *uav* and *gcs* are called as processes in the module *main*.

```
1     -- Define processes
2     uav : process uav(stateUav, stateGcs, isUavInHandoverRegion);
3     gcs1 : process gcs(1,commanderId, stateUav, stateGcs,
          ↪ isUavInHandoverRegion,
4     commanderHasSwitched);
5     gcs2 : process gcs(2,commanderId, stateUav, stateGcs,
          ↪ isUavInHandoverRegion,
6     commanderHasSwitched);
```

The module has a total of 6 variables:

- **commanderId:** Identifies the ID of the commander. Scale/Interpretation:
  - 1 = GCS A;
  - 2 = GCS B;
- **isUavInHandoverRegion:** Defines if the UAV is in the handover region. Scale Interpretation:
  - TRUE = UAV is in handover region;
  - FALSE = UAV is not in handover region;
- **stateUav:** Defines the state of the UAV. Scale/Interpretation:
  - idle = Idle mode, awaiting instructions;
  - receiveTelemetry = Receive telemetry information from UAV;
  - requestHandover = UAV is requesting handover to GCS Partner;
  - connectHandover = UAV connects to GCS Partner after it has been authorized for handover by GCS Commander.
- **stateGcs:** Defines the state of the GCS (Commander and Partner). Scale/Interpretation:
  - idle = Idle mode, awaiting instructions;
  - receiveTelemetry = Transmit telemetry information to GCS Commander;
  - acceptPartnerRequest = GCS Partner has received a handover request from UAV;
  - authorizeHandover = GCS Commander has authorized, after receiving information regarding a connection request from UAV to GCS Partner from GCS Partner;
  - droneConnectToPartner = GCS Partner connects to UAV after the UAV has been authorized to connect to the GCS.
  - disconnectCommander= GCS Commander disconnects from UAV after the UAV has connected to the GCS Partner and transfers authority to the GCS Partner.
- **commanderHasSwitched** - Auxiliary variable to inform the procedure has reached the final step. Scale/Intepretation:
  - TRUE = Commander has switched for the current handover procedure;
  - FALSE = Commander has not switched for current the handover procedure;
- **messageTimeout** - Defines if there has been a message timeout between two entities (Any combination of UAV/GCS Commander/GCS Partner);

Initialization of the module:

```
1     init(commanderId) := 1;
2     init(isUavInHandoverRegion) := TRUE;
3     init(stateUav) := idle;
4     init(stateGcs) := idle;
5     init(commanderHasSwitched) := FALSE;
6     init(messageTimeout) := FALSE;
```

Only the variables *isUavInHandoverRegion*, *commanderId* and *commanderHasSwitched* transition in the module *main* and the states of UAV and GCSs are updated in the respective modules.

```
1     next(isUavInHandoverRegion) :=
2         case
3             TRUE: {TRUE,FALSE};
4         esac;
```

The *isUavInHandoverRegion* transition can go TRUE or FALSE at every state. The reason for this is to evaluate how the model interacts if the UAV can move to or move away from the handover region.

```
1     next(commanderId) :=
2         case
3             commanderId = 1 & stateGcs = disconnectCommander & !
          ↪ commanderHasSwitched : 2;
4             commanderId = 2 & stateGcs = disconnectCommander & !
          ↪ commanderHasSwitched : 1;
5             TRUE: commanderId;
6         esac;
```

Whenever the commanderId is 1 and the GCS commander is in disconnectCommander state and the commanderHasSwitched is false, then it shall switch from 1 to 2. This can also happen in the inverse way.

```
1     next(commanderHasSwitched) :=
2         case
3             stateGcs = disconnectCommander : TRUE;
4             TRUE: FALSE;
5         esac;
```

Whenever the GCS Commander reaches the state disconnectCommander, it shall be TRUE. This is to guarantee that the switch only happens once per handover.

```
1     next(messageTimeout) :=
2         case
3             TRUE: {TRUE,FALSE};
4             --TRUE: TRUE;
5         esac;
```

Same application as isUavInHandoverRegion, the transition can go TRUE or FALSE at every state. The reason for this is to evaluate how the model interacts if the there is a timeout in the procedure between the entities.

Module uav

The module uav does not have any assigned variables as it uses the variables defined in the module main.
It only possesses a single state transition (stateUav):

```
1     next(stateUav) := case
2         stateUav = connectPartner & stateGcs = authorizeHandover &
            ↪ (isUavInHandoverRegion & !messageTimeout) :
            ↪ connectHandover;
3         stateUav = sendTelemetry & (isUavInHandoverRegion & !
            ↪ messageTimeout) : connectPartner;
4         stateUav = idle | stateGcs = receiveTelemetry:
            ↪ sendTelemetry;
5         stateUav = sendTelemetry | stateGcs = disconnectCommander |
            ↪  !isUavInHandoverRegion | messageTimeout : idle;
6         TRUE : stateUav;
7     esac;
```

The UAV shall initiate in state idle and transition only to sendTelemetry if the GCS Commander is expecting to receive the telemetry. This is an abstract way to model as in reality the UAV would constantly be sending the telemetry to the GCS Commander.
It shall transition to requestHandover, if the UAV is in sendTelemetry (to simulate latest telemetry result) and is in the handover region.
It shall await authorization of the handover from GCS Commander, and when the authorization is received, then transition to state connectHandover.
Finally, when the GCS Commander disconnects, it shall go to idle state, finishing the cycle.
In case the UAV is not in the handover region or a message timeout has occurred, then it shall transition to state idle.

Module gcs

The module gcs does not have any assigned variables as it uses the variables defined in the module main.
It only possesses a single state transition (stateGcs):

```
1     next(stateGcs) :=
2      case
3         stateGcs = droneConnectToPartner & (commanderId = id) & !
            ↪ commanderHasSwitched : disconnectCommander;
4         stateGcs = authorizeHandover & stateUav = connectHandover &
            ↪  !(commanderId = id) : droneConnectToPartner;
5         stateGcs = acceptPartnerRequest & (commanderId = id) :
            ↪ authorizeHandover;
6         stateGcs = receiveTelemetry & isUavInHandoverRegion & !(
            ↪ commanderId = id) & stateUav = requestHandover:
            ↪ acceptPartnerRequest;
7         stateGcs = idle & commanderId = id : receiveTelemetry;
8         stateGcs = receiveTelemetry | (stateGcs =
            ↪ disconnectCommander & commanderHasSwitched) : idle;
9         TRUE : stateGcs;
10     esac;
```

The GCS shall initiate in state idle and transition only to sendTelemetry if the GCS is the GCS Commander.
If the GCS is in state receiveTelemetry, the UAV is in handover region, the UAV is requesting handover and the GCS is not the commander (Partner), then it shall accept the partner request and go to state acceptPartnerRequest.

If the GCS is in state acceptPartnerRequest and it is the GCS Commander, then it shall transition to authorizeHandover. If the GCS is in state authorizeHandover, the UAV has requested to connect to the GCS Partner and the GCS is not the Commander (Partner), then it shall connect to the UAV as partner and transition to the state droneConnectToPartner. If the GCS is in state droneConnectToPartner, the GCS is GCS Commander and the Commander switch hasn't happened, then the GCS Commander shall disconnect from the UAV. In case the UAV is not in the handover region or a message timeout has occurred, then it shall transition to state idle.

Model Properties

```
1 LTLSPEC (stateUav = connectPartner & stateGcs =
        ↪ acceptPartnerRequest & isUavInHandoverRegion -> F
        ↪ stateGcs = authorizeHandover);
```

```
1 LTLSPEC (stateUav = connectPartner & stateGcs =
        ↪ acceptPartnerRequest & isUavInHandoverRegion -> F
        ↪ stateGcs = authorizeHandover);
```

This specification is used to verify that when the UAV is connecting to a partner and the GCS accepts the partner request while the UAV is in the handover region, the system will eventually authorize the handover. This ensures that the handover process completes when these conditions are met.

```
1 LTLSPEC (!isUavInHandoverRegion & commanderId = 1 & stateUav =
        ↪ connectPartner -> !isUavInHandoverRegion U (F ((
        ↪ commanderId = 2) & !isUavInHandoverRegion &
        ↪ commanderHasSwitched & stateGcs = disconnectCommander)
        ↪ ));
```

This specification verifies that a commander switch is possible outside of the handover region. It ensures that when the UAV connects to a partner while outside the handover region, the system will not switch the commander until the appropriate conditions are met and the commander has been successfully switched.

```
1 LTLSPEC (isUavInHandoverRegion & commanderId = 1 & stateUav =
        ↪ connectPartner & stateGcs = receiveTelemetry -> F (
        ↪ commanderId = 2));
```

This specification ensures that when the UAV is in the handover region, the handover process has started, and the GCS is receiving telemetry, the GCS commander will eventually be switched from Commander 1 to Commander 2. This guarantees that the handover process will result in a successful commander switch.

```
1 CTLSPEC EG (isUavInHandoverRegion & stateUav = idle -> EF (stateUav
        ↪  = connectPartner));
```

This specification checks that there is always a path where the UAV, while in the handover region and in the idle state, will eventually start the handover process by connecting to a partner. It ensures that the system can initiate the handover process under these conditions.

```
1 LTLSPEC G (stateUav = idle -> F stateUav = sendTelemetry);
```

This specification ensures that whenever the UAV is in the idle state, it will eventually send telemetry data. This guarantees that telemetry is consistently sent while the UAV is idle, ensuring continuous monitoring.

```
1 LTLSPEC (isUavInHandoverRegion & stateUav = connectPartner -> F
    ↪ stateUav = connectHandover);
```

This specification ensures that when the UAV is in the handover region and it connects to a partner, it will eventually complete the connection for handover. This guarantees that the UAV will follow through with the necessary steps for handover.

```
1 CTLSPEC (!isUavInHandoverRegion & stateUav = idle & stateGcs = idle
    ↪  -> EF commanderHasSwitched);
```

This specification checks that when both the UAV and GCS are in the idle state and the UAV is not in the handover region, there exists a path where the commander will eventually be switched. This ensures the system's ability to switch commanders under these conditions.

```
1 LTLSPEC (!isUavInHandoverRegion & stateUav = sendTelemetry -> F
    ↪ stateUav = requestAuthorizationFromCommander);
```

This specification ensures that when the UAV is not in the handover region and is sending telemetry, it will eventually request authorization from the commander. This ensures that proper authorization is sought in scenarios where the UAV is outside the handover region.

```
1 LTLSPEC (!isUavInHandoverRegion & stateUav = sendTelemetry &
    ↪ stateGcs = receiveTelemetry & !commanderHasSwitched ->
    ↪  F (stateUav = requestAuthorizationFromCommander & !
    ↪ isUavInHandoverRegion));
```

This specification ensures that when the UAV is not in the handover region, is sending telemetry, and the commander has not been switched, the system will eventually request authorization from the commander. This ensures that the handover process adheres to the necessary authorization steps.

```
1 LTLSPEC (stateUav = connectPartner & stateGcs =
    ↪ acceptPartnerRequest & isUavInHandoverRegion & !
    ↪ messageTimeout -> F stateGcs = authorizeHandover);
```

This specification ensures that when the GCS disconnects Commander 1, the switch has not occurred, and the UAV is in the handover region and there is no message timeout, the system will eventually switch to Commander 2. This ensures that the commander switch process is completed under these specific conditions.

```
1 CTLSPEC EG (stateGcs = idle & commanderId = 1 & !
    ↪ commanderHasSwitched -> EF commanderId = 2);
```

This specification checks that when the GCS is in the idle state, the commander is identified as Commander 1, and the switch has not occurred, there exists a path where the system will eventually switch to Commander 2. It verifies that the system supports the transition between commanders.

```
1 LTLSPEC G (stateGcs = disconnectCommander & commanderId = 2 & !
    ↪ commanderHasSwitched & isUavInHandoverRegion -> F
    ↪ commanderId = 1);
```

This specification ensures that when the GCS disconnects Commander 2, the switch has not occurred, and the UAV is in the handover region, the system will eventually switch back to Commander 1. This ensures the commander switch process can be reversed under these conditions.

```
1 LTLSPEC G (stateGcs = disconnectCommander & commanderId = 1 & !
          ↪ commanderHasSwitched -> F commanderId = 2 -> F(
          ↪ stateGcs = disconnectCommander & commanderId = 2 & !
          ↪ commanderHasSwitched) -> F(commanderId = 1));
```

This specification ensures that after disconnecting Commander 1, the system will switch to Commander 2 and then back to Commander 1, following a sequence of operations. It verifies the system's ability to handle multiple commander switches in a controlled manner.

```
1 CTLSPEC EG (stateGcs = idle & commanderId = 2 & !
          ↪ commanderHasSwitched -> EF commanderId = 1 -> EF
          ↪ commanderId = 2);
```

This specification checks that when the GCS is in the idle state, commanderId = 2, and the switch has not occurred, there exists a path where the system can switch to Commander 1 and then back to Commander 2. It ensures the system's capability to perform bidirectional commander switching.

```
1 CTLSPEC EG (stateUav = idle -> EF (stateUav = sendTelemetry));
```

This specification ensures that the UAV can transition from the idle state to the state where it sends telemetry. It checks the system's ability to reach the telemetry state from idle, ensuring the capability to monitor and communicate data.

```
1 CTLSPEC EG (stateUav = idle & isUavInHandoverRegion -> EF(stateUav
          ↪ = idle));
```

This specification ensures that the UAV can return to the idle state from any state while it is in the handover region. It verifies that the idle state is reachable under these conditions, ensuring the system's stability in the handover region.

```
1 CTLSPEC EG (stateUav = idle -> EF (stateUav = connectPartner));
```

This specification ensures that the UAV can transition from the idle state to the state where it connects to a partner. It checks that the handover process can be initiated from idle, ensuring the readiness of the system to connect when required.

```
1 CTLSPEC EG (stateUav = idle -> EF (stateUav = idle));
```

This specification checks that the UAV can remain in or return to the idle state from any other state. It verifies that the idle state is always accessible, ensuring the UAV's capability to reset its state.

```
1 CTLSPEC EG (stateUav = idle -> EF(stateUav = sendTelemetry));
```

This specification ensures that the UAV can send telemetry data from the idle state. It verifies that the telemetry transmission is reachable from idle, ensuring the UAV can communicate data regularly.

```
1 CTLSPEC EG (stateUav = idle -> EF (stateUav = connectHandover) ->
      ↪ EF (stateUav = idle));
```

This specification ensures that the UAV can initiate a handover connection from the idle state and then return to idle. It verifies the process of connecting for handover and returning to idle, ensuring the UAV's ability to complete and reset the handover process.

```
1 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = receiveTelemetry));
```

This specification checks that the GCS can transition from the idle state to the state where it receives telemetry. It verifies the GCS's ability to receive telemetry from idle, ensuring the GCS can handle incoming data.

```
1 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = acceptPartnerRequest))
      ↪ ;
```

This specification ensures that the GCS can transition from the idle state to the state where it accepts a partner request. It verifies the GCS's ability to engage in the handover process from idle, ensuring its readiness to accept connections.

```
1 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = authorizeHandover));
```

This specification ensures that the GCS can transition from the idle state to the state where it authorizes the handover. It verifies the GCS's capability to authorize the handover from idle, ensuring the handover process can be completed.

```
1 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = droneConnectToPartner)
      ↪ );
```

This specification ensures that the GCS can transition from the idle state to the state where it connects to the partner drone. It verifies the GCS's ability to establish a connection with the UAV, ensuring the handover process can be facilitated.

```
1 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = disconnectCommander)
      ↪ -> EF (stateGcs = idle));
```

This specification ensures that the GCS can transition from the idle state to the state where it disconnects the commander and then return to idle. It verifies the GCS's capability to disconnect and reset, ensuring the system's stability and flexibility.

5.3.3 Results

In this iteration, all requirements have at least one property successfully verified. The model incorporated the message timeout mechanic as well as the constant verification for the UAV in the handover region.

The message timeout is generic, as all entities will eventually timeout of the procedure, therefore it can be implemented as a simple boolean value. The UAV in handover region check is possible since the GCS and the UAV have access to the telemetry to determine if the UAV is the handover region.

The existing properties were modified to take into account the update of the model. All properties were proven to be consistent with the diagrams and the derived requirements for this iteration. The results demonstrate that the model effectively captures the intended functionality while remaining aligned with the specified requirements for iteration three.

New iterations are not necessary as no issue was found after implementing these changes.

Chapter 6

Conclusions

This dissertation has explored model checking in the context of UAV communication protocols, specifically focusing on the UAV authority transfer protocol presented in (Ferreira 2023). The research provides an examination of the sate of the art in model checking, delving into various models, tools and applications relevant to the protocol verification. Through this investigation, the importance of robustness and reliability in the communication protocols for UAV operations was highlighted.

The methodology proposed for evaluating the protocol using model checking represents a structured approach to verifying the robustness and reliability of communication protocol. By defining requirements for each entity involved in the operation, the research presents a baseline for the evaluation of the protocol.

During the iterative development process, at least one property was successfully verified for each requirement, demonstrating the functionality applied in the protocol. Initially, the model's complexity made it difficult to verify certain properties due to the state explosion through the use of many variables. In spite of the issue, two concerns were highlighted: the inability to cancel the handover procedure once it has begun and the lack of continuous verification of the UAV position inside the handover region.

In response to the state explosion issue, a subsequent iteration was introduced that aimed to simplify the model while focusing solely on the functionality. After both concerns were once again confirmed to exist, a third iteration was made to propose a solution to these concerns. A timeout/cancel operation was added to allow the procedure to return to an idle state, and the verification process was improved by constantly checking if the UAV was still in the handover region.

The final iteration successfully addressed all previously identified concerns, confirming that the model accurately captures the intended functionality. As a result, no additional iterations are required, as the model now meets all requirements and exhibits the desired behavior with no unresolved issues and can be used as a baseline for further analysis if needed.

Appendix A

Appendix A - Iteration 1 NuSMV Model

```
1  MODULE UavControl (commanderId, stateUav,
          ↪ varUav_connectionToGcsCommander,
          ↪ varGcs_isCommanderConnectionAuthorized,
          ↪ varUav_messageTypeToTransmitToCommander,
          ↪ varUav_messageTypeToTransmitToPartner,
          ↪ varUav_messageTypeToReceiveFromCommander,
          ↪ varUav_messageTypeToReceiveFromPartner,
          ↪ varUav_checkDistanceToGoal, commanderHasSwitched,
          ↪ keyboardPress)
2
3  FAIRNESS
4      -- Ensure that all processes have fair oportunities to execute
5      running;
6
7  -- The entry point for the main module
8  VAR
9    -- States
10   -- UAV
11   stateUAV: {poweredOff, startUp, operational}; -- Possible states
          ↪ for the UAV system.
12   stateUAV_telemetry:{notActive, connecting, telemetry}; --
          ↪ Possible states for the UAV system, while in operating
          ↪  state (this can be considered as a sub-state). In "
          ↪ Operating" state, the UAV can simultaneously execute
          ↪ actions, hence separating this into a group of states
          ↪ for a feature.
13   stateUAV_messageHandling: {listening, validating, executing}; --
          ↪ Defines the order of the single message execution.
          ↪ Only one message can be executed at a time.
14   stateUAV_handoverProcedure: {off, onGoing}; -- Possible states
          ↪ for the UAV system, specific to the handover procedure
          ↪ .
15   -- Variables UAV
16   varUav_isPowerOn: boolean; -- Defines if the UAV system is
          ↪ powered on. Scale/interpretation: TRUE=Powered On,
          ↪ FALSE=Powered Off;
17   varUav_isConfigurationLoaded: boolean; -- Defines if the
          ↪ configuration data is loaded into the UAV. Scale/
          ↪ interpretation: TRUE=Configuration data loaded, FALSE=
          ↪ Configuration data not loaded;
```

```
18    varUav_isConnectedToFlightController: boolean; -- Defines that
          ↪ the UAV is connected to the flight controller (for
          ↪ telemetry). Scale/interpretation: TRUE=Connected,
          ↪ FALSE=Not connected;
19    varUav_newReceivedMessage: boolean; -- Defines if there is an
          ↪ upcoming message from a GCS.
20    varUav_isReceivedMessageValid: boolean; -- Defines if a received
          ↪ messaged is valid. Scale/interpretation: TRUE=Valid,
          ↪ FALSE=Invalid;
21    varUav_messageType: {keyboardCommand, mission,
          ↪ handoverConnectToPartnerAsClient,
          ↪ handoverAuthorizationFromCommander}; -- Defines the
          ↪ message type. For more details, check the UAV diagram.
22    varUav_messageType_keyboardCommand: {Up, Down, Left, Right}; --
          ↪ Defines a basic movement for the UAV to execute.
23    varUav_performMovement: {Up, Down, Left, Right, None}; -- Defines
          ↪  a basic movement for the UAV to execute based on
          ↪ received message.
24    varUAV_movementFinished: boolean; -- Defines if the movement has
          ↪ finished executing. Scale/interpretation: TRUE=Valid,
          ↪ FALSE=Invalid;
25    varUav_isInHandoverRegion: boolean; -- Defines if the UAV is in
          ↪ handover region.  Scale/interpretation: TRUE=In
          ↪ handover region, FALSE=Not in handover region;
26    varUav_isGoalDistanceMeasurementEnabled: boolean; -- Defines if
          ↪ the distance is being check. Scale/interpretation:
          ↪ TRUE = Measuring; FALSE = Not Measuring.
27    varUav_sendConnectRequestToPartner: boolean; -- Defines the
          ↪ connection request to partner. Scale/interpretation:
          ↪ TRUE = Send request; FALSE = Do not send request.
28    varUav_retryConnectRequest: boolean; -- Defines the connect
          ↪ request to partner after failing at least once. Scale/
          ↪ Interpretation: TRUE = Retry send request; FALSE = Do
          ↪ not retry.
29    varUav_sendHandoverRequestToPartner: boolean; -- Defines the
          ↪ connect request as handover to partner. Scale/
          ↪ Interpretation: TRUE = Send handover request; FALSE =
          ↪ Do not send handover request.
30    varUav_retryHandoverRequestToPartner: boolean; -- Defines the
          ↪ connect requrest as handover after failing at least
          ↪ once. Scale/Interpretation: TRUE = Retry send request;
          ↪  FALSE = Do not retry.
31    varUav_currentGcsCommander: 1..2; -- Defines which GCS is the
          ↪ commander related to the UAV. Scale/Interpretation: 1
          ↪ = GCS A; 2 = GCS B.
32    -- Variables GCS
33    varGcs_isPowerOn: boolean; -- Defines if the GCS system is
          ↪ powered on. Scale/interpretation: TRUE=Powered On,
          ↪ FALSE=Powered Off;
34    varGcs_authorizationFromCommanderForHandover: boolean; --
          ↪ Defines the authorization to be sent from commander
          ↪ GCS to the UAV. Scale/interpretation: TRUE=Authorized,
          ↪  FALSE=Not Authorized;
```

```
35   varGcs_acknowledgeConnectMessage: boolean; -- Defines the
         ↪ acknowledgement of the connect request from GCS to the
         ↪  UAV. Scale/interpretation: TRUE = Acknowledged; FALSE
         ↪  = Not Acknowledged.
36   varGcs_acknowledgeHandoverMessage: boolean; -- Defines the
         ↪ acknowledgement of the handover request from GCS to
         ↪ the UAV. Scale/interpretation: TRUE = Acknowledged;
         ↪ FALSE = Not Acknowledged.
37
38 ASSIGN -- UAV
39   -- Initial state
40   --UAV
41   init(stateUAV) := poweredOff; -- The initial state is powered off
         ↪ . Whenever the UAV is powered on, it shall load the
         ↪ configuration. This way, the start-up process is
         ↪ equivalent in all scenarios.
42   init(stateUAV_telemetry) := notActive; -- The initial state is
         ↪ not active. It will only update when the UAV is in the
         ↪  "Operating" state.
43   init(stateUAV_messageHandling) := listening; -- The initial state
         ↪  is listening, as it will not be validating nor
         ↪ executing any messages during start-up.
44   init(stateUAV_handoverProcedure) := off; -- The initial state is
         ↪ off, as it will need actions to start the handover
         ↪ procedure.
45
46   --Default/start value for variables
47   -- UAV
48   init(varUav_isPowerOn) := TRUE;
49   init(varUav_isConfigurationLoaded) := FALSE;
50   init(varUAV_movementFinished) := FALSE;
51
52   -- State transitions
53   next(stateUAV) :=
54     case
55       stateUAV = poweredOff & varUav_isPowerOn = TRUE : startUp; --
         ↪  REQ-UAV-001: The UAV shall proceed to state "Start-up
         ↪ " when powered on.
56       stateUAV = startUp & varUav_isPowerOn = FALSE : poweredOff;
         ↪ -- REQ-UAV-002: If the UAV is in the "Start-up" state
         ↪ and is powered off, then it shall transition to "Power
         ↪  Off" state.
57       stateUAV = operational & varUav_isPowerOn = FALSE :
         ↪ poweredOff; -- REQ-UAV-003: If the UAV is in the "
         ↪ Operating" state and is powered off, then it shall
         ↪ transition to "Power Off" state.
58       stateUAV = startUp & varUav_connectionToGcsCommander =
         ↪ connected & varUav_isPowerOn = TRUE :  operational; --
         ↪  REQ-UAV-006: When the UAV connects to the configured
         ↪ Ground Control Station, it shall transition to "
         ↪ Operating" state.
59       TRUE : stateUAV;
60     esac;
61
```

```
62    next(stateUAV_telemetry) :=
63      case
64        stateUAV_telemetry = notActive & stateUAV = operational :
            ↪ connecting; -- REQ-UAV-007 - When the UAV transitions
            ↪ to "Operating" state, it shall connect to the flight
            ↪ controller. In addition, it shall keep retrying the
            ↪ connection request until it is successful.
65        stateUAV_telemetry = connecting &
            ↪ varUav_isConnectedToFlightController : telemetry; --
            ↪ REQ-UAV-008 - When the UAV is in "Operating" state and
            ↪  connected to the flight controller, it shall get
            ↪ current telemetry and transmit it to the ground
            ↪ control station.
66        !(stateUAV = operational) : notActive; -- No requirement, but
            ↪  it's implied. Perhaps I should add a requirement here
            ↪ . The goal is to reset the state when a power-off
            ↪ occurs.
67        TRUE: stateUAV_telemetry;
68      esac;
69
70    next(stateUAV_messageHandling) :=
71      case
72      stateUAV = operational & varUav_newReceivedMessage &
            ↪ stateUAV_messageHandling = listening : validating;
73      stateUAV = operational & varUav_newReceivedMessage &
            ↪ varUav_isReceivedMessageValid : executing;
74      stateUAV = operational & !varUav_newReceivedMessage : listening
            ↪ ; --REQ-UAV-009 - While the UAV transitions to "
            ↪ Operating" state, it shall listen continuously for
            ↪ incoming messages from ground control stations.
75      TRUE: stateUAV_messageHandling;
76      esac;
77
78    next(stateUAV_handoverProcedure) :=
79      case
80      stateUAV = operational & varUav_messageType =
            ↪ handoverConnectToPartnerAsClient : onGoing;
81      stateUAV = startUp : off;
82      TRUE: stateUAV_handoverProcedure;
83      esac;
84
85    -- Movement handling
86    next(varUav_performMovement) :=
87      case
88      -- REQ-UAV-012 - While the UAV is in "Operating" state, if the
            ↪ message type is "Keyboard Command", it shall perform a
            ↪  movement maneuver according to the content of the
            ↪ message
89      stateUAV = operational & varUav_isReceivedMessageValid &
            ↪ stateUAV_messageHandling = executing & keyboardPress =
            ↪  Up : Up;
90      stateUAV = operational & varUav_isReceivedMessageValid  &
            ↪ stateUAV_messageHandling = executing & keyboardPress =
            ↪  Down : Down;
```

```
91      stateUAV = operational & varUav_isReceivedMessageValid &
            ↪ stateUAV_messageHandling = executing & keyboardPress =
            ↪  Left : Left;
92      stateUAV = operational & varUav_isReceivedMessageValid &
            ↪ stateUAV_messageHandling = executing & keyboardPress =
            ↪  Right : Right;
93      TRUE: None;
94      esac;
95
96    -- Variable transitions
97    next(varUav_isConfigurationLoaded) :=
98      case
99        stateUAV = startUp : varUav_isConfigurationLoaded = TRUE; --
            ↪ REQ-UAV-004: When the UAV transitions to "Start-up"
            ↪ state, it shall load up the UAV parameters from the
            ↪ configuration data.
100       TRUE: varUav_isConfigurationLoaded;
101     esac;
102
103 next(varUav_connectionToGcsCommander) :=
104     case
105       stateUAV = startUp & varUav_isConfigurationLoaded &
            ↪ varUav_connectionToGcsCommander = noConnection:
            ↪ attemptingToConnect; -- REQ-UAV-005: When UAV is in "
            ↪ Start-up" state and has finished loading UAV
            ↪ parameters from configuration data, it shall connect
            ↪ to the configured Ground Control Station.
106       stateUAV = startUp & varUav_isConfigurationLoaded &
            ↪ varUav_connectionToGcsCommander = attemptingToConnect
            ↪ & varGcs_isCommanderConnectionAuthorized: connected;
107       TRUE: varUav_connectionToGcsCommander;
108     esac;
109
110
111   next(varUav_isConnectedToFlightController) :=
112     case
113       stateUAV_telemetry = connecting : {FALSE,TRUE}; -- Simulating
            ↪  50/50.
114       TRUE: varUav_isConnectedToFlightController;
115     esac;
116
117   next(varUav_isReceivedMessageValid) :=
118     case
119       stateUAV = operational & stateUAV_messageHandling =
            ↪ validating : {FALSE,TRUE}; -- (Simulating 50/50) REQ-
            ↪ UAV-010 - While the UAV is in "Operating" state, it
            ↪ shall validate any incoming messages from ground
            ↪ control stations.
120       stateUAV = operational & stateUAV_messageHandling = listening
            ↪  : FALSE; -- If it's listening, there is no current
            ↪ valid message. REQ-UAV-011 - While the UAV is in "
            ↪ Operating" state, it shall discard any invalid
            ↪ incoming messages from ground control stations. (This
            ↪ is done indirectly)
```

```
121       TRUE: varUav_isReceivedMessageValid;
122     esac;
123
124    next(varUav_isInHandoverRegion) :=
125     case
126     stateUAV = operational : {FALSE,TRUE}; -- Simulating 50/50 to
          ↪ make it possible to the UAV to leave the handover
          ↪ region at any given moment.
127     TRUE: varUav_isInHandoverRegion;
128     esac;
129
130   next(varUav_isGoalDistanceMeasurementEnabled) :=
131     case
132     stateUAV = operational & varUav_messageType = mission &
          ↪ varUav_isReceivedMessageValid : TRUE;
133     TRUE: FALSE;
134     esac;
135
136   next(varUav_checkDistanceToGoal) :=
137     case   varUav_isGoalDistanceMeasurementEnabled : {0..99}; --
          ↪ REQ-UAV-014 - While the UAV is in "Operating" state,
          ↪ if the message type is "Mission", it shall transmit
          ↪ distance to goal.
138     TRUE: varUav_checkDistanceToGoal;
139     esac;
140
141   next(varUav_sendConnectRequestToPartner) :=
142     case
143     -- REQ-UAV-015 - While the UAV is in "Operating" state, if the
          ↪ message type is "Handover - Connect to Partner as
          ↪ Client" and the UAV is within handover region, it
          ↪ shall send a message request to connect to partner. In
          ↪  addition, it shall retry to transmit the message in
          ↪ case of failure.
144     stateUAV = operational & varUav_messageType =
          ↪ handoverConnectToPartnerAsClient &
          ↪ varUav_isReceivedMessageValid &
          ↪ varUav_isInHandoverRegion  : TRUE;
145     TRUE : FALSE;
146     esac;
147
148   next(varUav_retryConnectRequest) := case
149     -- REQ-UAV-015 - While the UAV is in "Operating" state, if the
          ↪ message type is "Handover - Connect to Partner as
          ↪ Client" and the UAV is within handover region, it
          ↪ shall send a message request to connect to partner. In
          ↪  addition, it shall retry to transmit the message in
          ↪ case of failure.
150     stateUAV = operational & varUav_messageType =
          ↪ handoverConnectToPartnerAsClient &
          ↪ varUav_isInHandoverRegion &
          ↪ varUav_sendConnectRequestToPartner & !
          ↪ varGcs_acknowledgeConnectMessage : TRUE;
151     TRUE : FALSE;
```

```
152     esac;
153
154     next(varUav_sendHandoverRequestToPartner) := case
155       -- REQ-UAV-016 - While the UAV is in "Operating" state, after
              ↪ successfully transmitting the message request to
              ↪ connect to partner, it shall transmit a handover
              ↪ request to partner. In addition, it shall retry to
              ↪ transmit the message in case of failure.
156       stateUAV = operational & varGcs_acknowledgeConnectMessage :
              ↪ TRUE;
157       TRUE : FALSE;
158     esac;
159
160     next(varUav_retryHandoverRequestToPartner) := case
161       -- REQ-UAV-016 - While the UAV is in "Operating" state, after
              ↪ successfully transmitting the message request to
              ↪ connect to partner, it shall transmit a handover
              ↪ request to partner. In addition, it shall retry to
              ↪ transmit the message in case of failure.
162       stateUAV = operational & varUav_messageType =
              ↪ handoverConnectToPartnerAsClient &
              ↪ varUav_sendHandoverRequestToPartner & !
              ↪ varGcs_acknowledgeHandoverMessage : TRUE;
163       TRUE : FALSE;
164     esac;
165
166     next(varUav_currentGcsCommander) := case
167       -- REQ-UAV-017 - While the UAV is in "Operating" state, after
              ↪ successfully transmitting the connect handover request
              ↪  to partner and disconnect message from commander, it
              ↪ shall change commander to current partner.
168       stateUAV = operational &
              ↪ varGcs_authorizationFromCommanderForHandover &
              ↪ varUav_currentGcsCommander = 1 : 2;
169       stateUAV = operational &
              ↪ varGcs_authorizationFromCommanderForHandover &
              ↪ varUav_currentGcsCommander = 2 : 1;
170       TRUE : varUav_currentGcsCommander;
171     esac;
172   -- SPECS
173
174   -- ### UAV ###
175
176   --REQ-UAV-001: The UAV shall proceed to state "Start-up" when
              ↪ powered on.
177   CTLSPEC ((stateUAV = poweredOff & varUav_isPowerOn) -> EF(stateUAV
              ↪ = startUp)) -- If the UAV is powered on implies that
              ↪ it eventually will reach "Start-up" state.
178
179   -- REQ-UAV-002: If the UAV is in the "Start-up" state and is
              ↪ powered off, then it shall transition to "Power Off"
              ↪ state.
```

```
180 LTLSPEC ((stateUAV = startUp) -> ((varUav_isPowerOn & !next(varUav_isPowerOn)) -> F(stateUAV = poweredOff))) -- If the UAV is powered off while in the "Start-up" state implies that it will eventually reach the "Powered Off" state.
181 -- Would like to use X instead of F, but since the transition to "Start-up" state is right after "Powered Off" state and requires the same variable to trasition, NuSMV accuses this LTLSPEC as false as it can get set the "Start-up" and varUav_isPowerOn = FALSE.
182
183 -- REQ-UAV-003: If the UAV is in the "Operating" state and is powered off, then it shall transition to "Power Off" state.
184 LTLSPEC G((stateUAV = operational) -> ((varUav_isPowerOn & !next(varUav_isPowerOn)) -> X(stateUAV = poweredOff))) -- Globally, if the UAV is powered off while in the "Operational" state implies that the next state will be the "Powered Off" state.
185
186  -- REQ-UAV-006: When the UAV connects to the configured Ground Control Station, it shall transition to "Operating" state.
187  LTLSPEC G((varUav_connectionToGcsCommander = connected) -> F(stateUAV = operational)) -- Globally, if the UAV is connected to the GCS Commander, then it will eventually reach "Operational" state.
188  LTLSPEC G((varUav_isPowerOn & varUav_isConfigurationLoaded & varUav_connectionToGcsCommander = connected) -> F(stateUAV = operational)) -- Globally, if there is power, the configuration is loaded and the UAV is connected to the GCS, eventually it will reach "Operational" state. This is another variation but with more specification on the value of the variables.
189
190 -- REQ-UAV-007 - When the UAV transitions to "Operating" state, it shall connect to the flight controller. In addition, it shall keep retrying the connection request until it is successful.
191 LTLSPEC G((stateUAV = operational) -> F(stateUAV_telemetry = connecting)) -- Globally, if the UAV is in the "Operational" state, then UAV shall eventually try to connect to the flight controller.
192
193 --REQ-UAV-008 - When the UAV is in "Operating" state and connected to the flight controller, it shall get current telemetry and transmit it to the ground control station.
194 LTLSPEC G((stateUAV = operational) -> F(stateUAV_telemetry = telemetry)) -- Globally, if the UAV is in the "Operational" state, then UAV shall eventually be in "Telemetry" sub-state.
195
```

```
196 -- Regardless of what I do, this statement is always true, even
         ↪ when adding ! to the third argument.
197 LTLSPEC G((stateUAV_handoverProcedure = onGoing) -> (stateUAV =
         ↪ operational -> stateUAV_handoverProcedure != off)) --
         ↪ Globally, whenever the handover procedure is ongoing,
         ↪ if the UAV is operational, then the handover procedure
         ↪  cannot be off
198
199 -- REQ-UAV-009 - While the UAV transitions to "Operating" state, it
         ↪  shall listen continuously for incoming messages from
         ↪ ground control stations.
200 LTLSPEC G (stateUAV = startUp & next(stateUAV) = operational ->
         ↪ stateUAV_messageHandling = listening) -- Globally, if
         ↪ the UAV transitions from state Start-up to Operational
         ↪ , the UAV shall be listening for incoming messages.
201
202 -- REQ-UAV-010 - While the UAV is in "Operating" state, it shall
         ↪ validate any incoming messages from ground control
         ↪ stations
203 CTLSPEC AG (stateUAV = operational & varUav_newReceivedMessage ->
         ↪ AX (stateUAV_messageHandling = validating)) --  For
         ↪ all paths, if the UAV is operational and there is a
         ↪ new received message, it should always be in the
         ↪ validating state for message handling.
204
205 -- REQ-UAV-011 - While the UAV is in "Operating" state, it shall
         ↪ discard any invalid incoming messages from ground
         ↪ control stations.
206 LTLSPEC G (stateUAV = operational & varUav_newReceivedMessage & !
         ↪ varUav_isReceivedMessageValid ->
         ↪ stateUAV_messageHandling = listening) -- Globally, if
         ↪ the UAV is opertioanl and there is a new received
         ↪ message that is invalid, it should always be in the
         ↪ listening for message handling
207 -- It is an implicit check, as there is no indication on discarded
         ↪ messages. Following this logic, it indicates that the
         ↪ message is discarded and the UAV is ready to listen
         ↪ for new messages.
208
209 -- REQ-UAV-012 - While the UAV is in "Operating" state, if the
         ↪ message type is "Keyboard Command", it shall perform a
         ↪  movement maneuver according to the content of the
         ↪ message.
210 CTLSPEC EX ((stateUAV = operational & varUav_isReceivedMessageValid
         ↪  & varUav_messageType = keyboardCommand) -> (
         ↪ varUav_performMovement = Up | varUav_performMovement =
         ↪  Down | varUav_performMovement = Left |
         ↪ varUav_performMovement = Right)) -- If the UAV is in
         ↪ the operational state, has received a valid message,
         ↪ and the message type is a keyboard command, then in
         ↪ the next state, the UAV must perform a movement
         ↪ command (Up, Down, Left, or Right).
211
```

```
212 -- REQ-UAV-013 - While the UAV is in "Operating" state, if the
          ↪ message type is "Mission", it shall verify the
          ↪ geofences.
213 -- No check nor code implemented as there is no definition or
          ↪ explanation to what is a geofence and how they are
          ↪ verified.
214
215 -- REQ-UAV-014 - While the UAV is in "Operating" state, if the
          ↪ message type is "Mission", it shall set distance to
          ↪ goal.
216 CTLSPEC AG (stateUAV = operational & varUav_messageType = mission &
          ↪  varUav_isReceivedMessageValid -> AX (
          ↪ varUav_isGoalDistanceMeasurementEnabled)) -- Whenever
          ↪ the UAV is operational, the message type is Mission
          ↪ and valid, it should always set the distance to the
          ↪ goal.
217 CTLSPEC AG (stateUAV = operational & varUav_messageType = mission &
          ↪  varUav_isGoalDistanceMeasurementEnabled -> AX (
          ↪ varUav_checkDistanceToGoal >=0 &
          ↪ varUav_checkDistanceToGoal <=99  )) -- Whenever the
          ↪ UAV is operational, the message type is mission and
          ↪ the goal measurement process is enabled, the possible
          ↪ range to obtain is 0 to 99 meters.
218
219 -- REQ-UAV-015 - While the UAV is in "Operating" state, if the
          ↪ message type is "Handover - Connect to Partner as
          ↪ Client" and the UAV is within handover region, it
          ↪ shall send a message request to connect to partner. In
          ↪  addition, it shall retry to transmit the message in
          ↪ case of failure.
220 LTLSPEC G (stateUAV = operational & varUav_messageType =
          ↪ handoverConnectToPartnerAsClient &
          ↪ varUav_isInHandoverRegion -> (
          ↪ varUav_sendConnectRequestToPartner & X (
          ↪ varUav_sendConnectRequestToPartner |
          ↪ varUav_retryConnectRequest))) -- Whenever the UAV is
          ↪ in the operational, the message type is
          ↪ handoverConnectToPartnerAsClient, and the UAV is
          ↪ within the handover region, it should send a message
          ↪ request to connect to the partner or, it should retry
          ↪ in case of failure.
221
222 -- REQ-UAV-016 - While the UAV is in "Operating" state, after
          ↪ successfully transmitting the message request to
          ↪ connect to partner, it shall transmit a handover
          ↪ request to partner. In addition, it shall retry to
          ↪ transmit the message in case of failure.
223 LTLSPEC G (stateUAV = operational &
          ↪ varGcs_acknowledgeConnectMessage -> (
          ↪ varUav_sendHandoverRequestToPartner & X (
          ↪ varUav_sendHandoverRequestToPartner |
          ↪ varUav_retryHandoverRequestToPartner))
224
```

```
225 -- REQ-UAV-017 - While the UAV is in "Operating" state, after
        ↪ successfully transmitting the connect handover request
        ↪  to partner and disconnect message from commander, it
        ↪ shall change commander to current partner.
226 LTLSPEC G (stateUAV = operational &
        ↪ varUav_sendHandoverRequestToPartner &
        ↪ varGcs_authorizationFromCommanderForHandover &
        ↪ varUav_currentGcsCommander = 1 -> next(
        ↪ varUav_currentGcsCommander) != 1)
227
228 --LTLSPEC -- this spec is not working as intended. (T)
229  -- LTLSPEC G((stateUAV_handoverProcedure = onGoing) -> ((stateUAV
        ↪ = operational & (stateUAV_handoverProcedure != off)))
        ↪  -- Globally, whenever the UAV handover procedure is
        ↪ ongoing, it must always be true (globally) that if the
        ↪  UAV is operational, then the UAV handover procedure
        ↪ cannot be off.
230   LTLSPEC G((stateUAV_handoverProcedure = onGoing) -> (stateUAV =
        ↪ operational -> stateUAV_handoverProcedure != off))
231
232   --CTLSPEC AG (stateUAV_handoverProcedure = onGoing -> AF stateUAV
        ↪  = poweredOff)
233
234 -- The CTLSPEC below proves that there is no way to stop the
        ↪ handoverProcedure without turning off the UAV. The
        ↪ CTLSPEC will fail.
235  --CTLSPEC EF (stateUAV_handoverProcedure = onGoing & AX (
        ↪ stateUAV_handoverProcedure = off & stateUAV !=
        ↪ poweredOff))
236
237 MODULE GcsControl (id, commanderId, staceGcs,
        ↪ varUav_connectionToGcsCommander,
        ↪ varGcs_isCommanderConnectionAuthorized,
        ↪ varGcs_messageTypeCommanderToReceive,
        ↪ varGcs_messageTypePartnerToReceive,
        ↪ varGcs_messageTypeCommanderToTransmit,
        ↪ varGcs_messageTypePartnerToTransmit,
        ↪ varUav_checkDistanceToGoal,varGcs_isPartnerConnected,
        ↪ commanderHasSwitched, keyboardPress)
238
239 FAIRNESS
240     -- Ensure that all processes have fair oportunities to execute
241     running;
242 VAR
243 -- States
244 stateGcs: {poweredOff, startUp, operational}; -- Possible states
        ↪ for the GCS system.
245
246 --Variables
247 varGcs_isPowerOn: boolean; -- Defines if the GCS is powered on.
        ↪ Scale/Interpretation: TRUE = Power is Active; FALSE =
        ↪ Power is Inactive.
```

```
248 varGcs_isConfigurationLoaded: boolean; -- Defines if the
        ↪ configuration has been loaded in the GCS. Scale/
        ↪ Interpretation: TRUE = Configuration is loaded; FALSE
        ↪ = Configuration is not loaded.
249 stateGcs_messageHandling: {listening, validating, executing}; --
        ↪ Defines the order of the single message execution.
        ↪ Only one message can be executed at a time. Scale/
        ↪ Interpretation: listening = Listening for upcoming
        ↪ messages, validating: Verifying if the received
        ↪ message is valid, executing: Executing a process based
        ↪  on the content of the message.
250 varGcs_newReceivedMessage: boolean; -- Defines if a new message has
        ↪  been received from the UAV. Scale/Interpretation:
        ↪ TRUE = New message, FALSE = No new message;
251 varGcs_isNewMessageValid: boolean; -- Defines if the new message is
        ↪  valid. Scale/Interpretation: TRUE = Valid, FALSE =
        ↪ Invalid.
252 varGcs_displayLatestTelemetry: boolean; -- Defines if the GCS
        ↪ commander displays the latest telemetry transmitted
        ↪ from the UAV. Scale/Interpretation: TRUE = Displaying,
        ↪  FALSE = Not displaying.
253 varGcs_isCheckRouteConfigured: boolean; -- Defines if the check
        ↪ route procedure is configured. Scale/Interpretation:
        ↪ TRUE = Configured, FALSE = Not configured.
254 varGcs_isUavOnRoute: boolean; -- Defines if UAV is on the correct
        ↪ route. Scale/Interpretation: TRUE = On correct route,
        ↪ FALSE = On incorrect route.
255 varGcs_missionDistance: 0..99; -- Defines the distance to goal as
        ↪ an integer. Scale/interpretation: 1=1 meter; Range =
        ↪ 0..99 meters.
256
257 ASSIGN -- GCS
258   init(stateGcs) := poweredOff;
259   init(varGcs_isConfigurationLoaded) := FALSE;
260   init(stateGcs_messageHandling) := listening;
261   init(varGcs_newReceivedMessage) := FALSE;
262   init(varGcs_isNewMessageValid) := FALSE;
263
264
265 next(stateGcs) :=
266    case
267      stateGcs = poweredOff & varGcs_isPowerOn : startUp; -- REQ-GCS
        ↪ -001 - The GCS shall proceed to state "Start-up" when
        ↪ powered on.
268      stateGcs = startUp & !varGcs_isPowerOn : poweredOff; -- REQ-
        ↪ GCS-002 - If the GCS is in the "Start-up" state and is
        ↪  powered off, then it shall transition to "Power Off"
        ↪ state.
269      stateGcs = operational & !varGcs_isPowerOn : poweredOff; --
        ↪ REQ-GCS-003 - If the GCS is in the "Operating" state
        ↪ and is powered off, then it shall transition to "Power
        ↪  Off" state.
```

```
270       stateGcs = startUp & varGcs_isPowerOn &
            ↪ varGcs_isConfigurationLoaded &
            ↪ varUav_connectionToGcsCommander = connected:
            ↪ operational; -- REQ-GCS-006 - When the GCS has
            ↪ connected to the configured UAV, it shall transition
            ↪ to "Operating" state.
271       TRUE : stateGcs;
272     esac;
273
274 -- Variable transitions
275  next(varGcs_isConfigurationLoaded) :=
276     case
277       stateGcs = startUp : varGcs_isConfigurationLoaded = TRUE; --
            ↪ REQ-UAV-004: When the UAV transitions to "Start-up"
            ↪ state, it shall load up the UAV parameters from the
            ↪ configuration data.
278       stateGcs = operational : varGcs_isConfigurationLoaded = TRUE;
279       stateGcs = poweredOff : varGcs_isConfigurationLoaded = FALSE;
280       TRUE: varGcs_isConfigurationLoaded;
281     esac;
282
283   next(varGcs_isCommanderConnectionAuthorized) :=
284     case
285       stateGcs = startUp & varGcs_isConfigurationLoaded & id =
            ↪ commanderId: TRUE; -- REQ-GCS-005 - When the GCS is in
            ↪  "Start-up" state and has loaded configuration data,
            ↪ it shall connect to the configured UAV.
286       stateGcs = poweredOff & id = commanderId: FALSE;
287       TRUE: varGcs_isCommanderConnectionAuthorized;
288     esac;
289
290   next(stateGcs_messageHandling) :=
291     case
292       stateGcs = operational & varGcs_newReceivedMessage &
            ↪ stateGcs_messageHandling = listening : validating; --
            ↪ REQ-GCS-008 - While the GCS is in "Operating" state,
            ↪ it shall validate any incoming messages from the UAV.
293       stateGcs = operational & varGcs_newReceivedMessage &
            ↪ stateGcs_messageHandling = validating &
            ↪ varGcs_isNewMessageValid : executing; -- REQ-GCS-009 -
            ↪  While the GCS is in "Operating" state, it shall
            ↪ discard any invalid incoming messages from the UAV.
294       stateGcs = operational & !varGcs_newReceivedMessage :
            ↪ listening; -- REQ-GCS-007 - When the GCS transitions
            ↪ to "Operating" state, it shall continuously listen to
            ↪ incoming messages.
295       TRUE: stateGcs_messageHandling;
296     esac;
297
298   next(varGcs_isNewMessageValid) :=
299     case
300     -- REQ-GCS-008 - While the GCS is in "Operating" state, it
            ↪ shall validate any incoming messages from the UAV.
```

```
301        varGcs_newReceivedMessage : {TRUE,FALSE}; -- Simulating valid
               ↪  and invalid messages when a new message is received.
302        !varGcs_newReceivedMessage : FALSE;
303        TRUE: varGcs_isNewMessageValid;
304     esac;
305
306    next(varGcs_newReceivedMessage) :=
307      case
308      -- REQ-GCS-008 - While the GCS is in "Operating" state, it
               ↪ shall validate any incoming messages from the UAV.
309        TRUE: {TRUE,FALSE};
310      esac;
311
312    next(varGcs_displayLatestTelemetry) :=
313      case
314        id = commanderId & varGcs_messageTypeCommanderToReceive =
               ↪ telemetry & stateGcs_messageHandling = executing :
               ↪ TRUE; -- If the received telemetry message is valid,
               ↪ then display the latest information in the GCS
               ↪ commander.
315        TRUE: FALSE;
316      esac;
317
318    next(varGcs_isUavOnRoute) :=
319      case
320        stateGcs = operational & varGcs_displayLatestTelemetry : {
               ↪ TRUE,FALSE}; -- REQ-GCS-011 - While the GCS is in "
               ↪ Operating" state and "check route" is configured, it
               ↪ shall analyse latest telemetry to verify if the UAV is
               ↪  currently on route.
321        TRUE: TRUE;
322      esac;
323
324    next(varGcs_missionDistance) :=
325      case
326        stateGcs = operational & varGcs_messageTypeCommanderToReceive
               ↪  = mission : varUav_checkDistanceToGoal;
327        TRUE: varGcs_missionDistance;
328      esac;
329
330    next(varGcs_messageTypePartnerToTransmit) :=
331    case
332      stateGcs = operational & varGcs_messageTypePartnerToReceive =
               ↪ handoverRequest & varGcs_isNewMessageValid & !(id =
               ↪ commanderId): acknowledgeConnectMessage;
333      TRUE: none;
334    esac;
335
336    next(varGcs_messageTypeCommanderToTransmit) :=
337    case
338      stateGcs = operational & varGcs_messageTypeCommanderToReceive =
               ↪  onHandover & varGcs_isNewMessageValid & (id =
               ↪ commanderId): okHandover;
```

```
339     stateGcs = operational & !(keyboardPress = none) & (id =
            ↪ commanderId): keyboardCommand;
340     TRUE: none;
341   esac;
342
343 -- ### GCS SPECS ###
344
345 -- REQ-GCS-001 - The GCS shall proceed to state "Start-up" when
            ↪ powered on.
346 CTLSPEC ((stateGcs = poweredOff & varGcs_isPowerOn) -> EF(stateGcs
            ↪ = startUp)) -- If the UAV is powered on implies that
            ↪ it eventually will reach "Start-up" state.
347
348 -- REQ-GCS-002 - If the GCS is in the "Start-up" state and is
            ↪ powered off, then it shall transition to "Power Off"
            ↪ state.
349 LTLSPEC ((stateGcs = startUp) -> ((varGcs_isPowerOn & !next(
            ↪ varGcs_isPowerOn)) -> F(stateGcs = poweredOff))) -- If
            ↪  the GCS is powered off while in the "Start-up" state
            ↪ implies th\ at it will eventually reach the "Powered
            ↪ Off" state.
350
351 -- REQ-GCS-003 - If the GCS is in the "Operating" state and is
            ↪ powered off, then it shall transition to "Power Off"
            ↪ state.
352 LTLSPEC ((stateGcs = operational) & !varGcs_isPowerOn -> next(
            ↪ stateGcs = poweredOff) U varGcs_isPowerOn) -- If the
            ↪ GCS is operational and is not powered, then the next
            ↪ state will be powered off until it is powered on again
            ↪ .
353
354 -- REQ-GCS-004 - When the GCS transitions to "Start-up" state, it
            ↪ shall load up the parameters from the configuration
            ↪ data.
355 LTLSPEC G((varGcs_isPowerOn & varGcs_isConfigurationLoaded &
            ↪ varUav_connectionToGcsCommander = connected) -> next(
            ↪ stateGcs = operational) U (varGcs_isPowerOn &
            ↪ varGcs_isConfigurationLoaded &
            ↪ varUav_connectionToGcsCommander = connected)) --
            ↪ Globally, if there is power, the configuration is
            ↪ loaded , eventually it will reach "Operational" state,
            ↪  assuming the power is on and the configuration is
            ↪ loaded.
356
357 -- REQ-GCS-005 - When the GCS is in "Start-up" state and has loaded
            ↪  configuration data, it shall connect to the
            ↪ configured UAV.
```

```
358 LTLSPEC (stateGcs = startUp & varGcs_isConfigurationLoaded &
        ↪ varUav_connectionToGcsCommander = attemptingToConnect
        ↪ & varGcs_isCommanderConnectionAuthorized & id =
        ↪ commanderId -> X(varUav_connectionToGcsCommander =
        ↪ connected)) -- If the GCS Commander is in state start-
        ↪ up, has the configuration loaded, has authorized the
        ↪ uav connection and the uav is attempting to connect to
        ↪  the commander GCS, then the GCS shall go into state
        ↪ operational and the UAV shall be connected to the GCS
        ↪ Commander.
359
360 -- REQ-GCS-006 - When the GCS has connected to the configured UAV,
        ↪ it shall transition to "Operating" state.
361 LTLSPEC (stateGcs = startUp & varGcs_isConfigurationLoaded &
        ↪ varUav_connectionToGcsCommander = attemptingToConnect
        ↪ & varGcs_isCommanderConnectionAuthorized & id =
        ↪ commanderId -> X(stateGcs = operational &
        ↪ varUav_connectionToGcsCommander = connected)) -- If
        ↪ the GCS Commander is in state start-up, has the
        ↪ configuration loaded, has authorized the uav
        ↪ connection and the uav is attempting to connect to the
        ↪  commander GCS, then the GCS shall go into state
        ↪ operational and the UAV shall be connected to the GCS
        ↪ Commander.
362
363 -- REQ-GCS-007 - When the GCS transitions to "Operating" state, it
        ↪ shall continuously listen to incoming messages.
364 LTLSPEC (stateGcs = operational -> X(stateGcs_messageHandling =
        ↪ listening)) -- If the GCS is operational, eventually
        ↪ it will be listening for upcoming messages.
365
366 -- REQ-GCS-008 - While the GCS is in "Operating" state, it shall
        ↪ validate any incoming messages from the UAV.
367 LTLSPEC G(stateGcs = operational & varGcs_newReceivedMessage &
        ↪ stateGcs_messageHandling = listening -> (
        ↪ stateGcs_messageHandling = validating)) -- Globally,
        ↪ if the GCS is operational and a new message is
        ↪ received, it shall always be validated.
368
369 -- REQ-GCS-009 - While the GCS is in "Operating" state, it shall
        ↪ discard any invalid incoming messages from the UAV.
370 LTLSPEC G(stateGcs = operational & varGcs_newReceivedMessage &
        ↪ stateGcs_messageHandling = validating & !
        ↪ varGcs_isNewMessageValid -> !(stateGcs_messageHandling
        ↪  = executing) U (stateGcs_messageHandling = validating
        ↪  & varGcs_isNewMessageValid)) -- Globally, if the GCS
        ↪ is operational, a new invalid message is received,
        ↪ then it will never execute until a new valid message
        ↪ is received.
371
372 -- REQ-GCS-010 - While the GCS is in "Operating" state and received
        ↪  message type is "Telemetry", it shall display latest
        ↪ telemetry data.
```

```
373 LTLSPEC G(stateGcs = operational & stateGcs_messageHandling =
        ↪ executing & id = commanderId &
        ↪ varGcs_messageTypeCommanderToReceive = telemetry ->
        ↪ varGcs_displayLatestTelemetry) -- Globally, if the GCS
        ↪  Commander is operational and a valid telemetry is
        ↪ received, then it shall display the latest telemetry
        ↪ information.
374
375 -- REQ-GCS-011 - While the GCS is in "Operating" state and "check
        ↪ route" is configured, it shall analyse latest
        ↪ telemetry to verify if the UAV is currently on route.
376 LTLSPEC G(stateGcs = operational & varGcs_isCheckRouteConfigured &
        ↪ varGcs_displayLatestTelemetry -> varGcs_isUavOnRoute)
        ↪ -- If the GCS has the check route configured and the
        ↪ latest telemetry has been displayed then the route
        ↪ shall be checked.
377
378 -- REQ-GCS-012 - While the GCS is in "Operating" state and received
        ↪  message type is "Connect", it shall setup the UAV
        ↪ connection and transmit an acknowledgement message to
        ↪ the UAV upon a successful connection setup
379 LTLSPEC (stateGcs = startUp & varGcs_isConfigurationLoaded &
        ↪ varUav_connectionToGcsCommander = attemptingToConnect
        ↪ & varGcs_isCommanderConnectionAuthorized & id =
        ↪ commanderId -> X(stateGcs = operational &
        ↪ varUav_connectionToGcsCommander = connected)) -- If
        ↪ the GCS Commander is in state start-up, has the
        ↪ configuration loaded, has authorized the uav
        ↪ connection and the uav is attempting to connect to the
        ↪  commander GCS, then the GCS shall go into state
        ↪ operational and the UAV shall be connected to the GCS
        ↪ Commander.
380
381 -- REQ-GCS-013 - While the GCS is in "Operating" state and received
        ↪  message type is "Mission", it shall load distance to
        ↪ mission goal from UAV.
382 LTLSPEC (stateGcs = operational &
        ↪ varGcs_messageTypeCommanderToReceive = mission &
        ↪ varUav_checkDistanceToGoal = 10 ->
        ↪ varGcs_missionDistance = 10)
383
384 -- REQ-GCS-014 - While the GCS is in "Operating" state and received
        ↪  message type is "Handover Request", it shall transmit
        ↪  an acknowledgement message to the UAV.
385 LTLSPEC (stateGcs = operational &
        ↪ varGcs_messageTypePartnerToReceive = handoverRequest &
        ↪  !(id = commanderId) & varGcs_isNewMessageValid ->
        ↪ varGcs_messageTypePartnerToTransmit =
        ↪ acknowledgeConnectMessage)
386
387 -- REQ-GCS-015 - While the GCS is in "Operating" state and received
        ↪  message type is "On Handover", it shall transmit an "
        ↪ OK Handover" to the UAV.
```

```
388 LTLSPEC (stateGcs = operational &
        ↪ varGcs_messageTypeCommanderToReceive = onHandover & !(
        ↪ id = commanderId) & varGcs_isNewMessageValid ->
        ↪ varGcs_messageTypeCommanderToTransmit = okHandover)
389
390 -- REQ-GCS-016 - While the GCS is in "Operating" state and received
        ↪  message type is "Connect Handover", it shall set UAV
        ↪ communication as client and transmit a "UAV Connected"
        ↪  to the UAV.
391 LTLSPEC (stateGcs = operational &
        ↪ varGcs_messageTypePartnerToReceive = connectHandover
        ↪ -> varGcs_isPartnerConnected)
392
393 -- REQ-GCS-018 - When the GCS transitions to "Operating" state and
        ↪ it detects a keyboard command, it shall transmit a
        ↪ validated command to the UAV.
394 LTLSPEC (stateGcs = operational & !(keyboardPress = none) ->
        ↪ varGcs_messageTypeCommanderToTransmit =
        ↪ keyboardCommand)
395
396  MODULE keyboardInput (keyboardPress) -- To generate random key
        ↪ presses.
397
398  ASSIGN
399
400   next(keyboardPress) :=
401     case
402       TRUE: {none, left, right, up, down}; -- Random keyboard press
        ↪  simulation.
403     esac;
404
405 MODULE
406     main
407
408 VAR
409 commanderId: 1..2; -- Definition of the commander Id. Scale/
        ↪ Interpretation: The number represents the ID of the
        ↪ GCS in control.
410 varUav_connectionToGcsCommander:  {noConnection,
        ↪ attemptingToConnect, connected}; -- Defines that the
        ↪ UAV is connected to a GCS (commander). Scale/
        ↪ interpretation: noConnection = No connection available
        ↪ ; attemptingToConnect = Has requested connection to
        ↪ the GCS, connected = Is currently connected to the GCS
        ↪  commander.
411 varGcs_isPartnerConnected: boolean; -- Defines if the partner GCS
        ↪ is connected to the UAV at the end of the handover
        ↪ procedure.
412 varGcs_isCommanderConnectionAuthorized: boolean; -- Defines the
        ↪ authroization for a start-up connection for the UAV to
        ↪  connect to the GCS commander. Scale/Interpretation:
        ↪ TRUE = Authorized connection, FALSE = Not authorized;
```

```
413 messageFromUavToGcsCommander: {none, telemetry, mission, onHandover
        ↪ }; -- Defines the possible message types to send from
        ↪ UAV to the GCS.
414 messageFromUavToGcsPartner: {none, handoverRequest, connectHandover
        ↪ }; -- Defines the possible message types to send from
        ↪ UAV to the GCS.
415 messageFromGcsCommanderToUav: {none, telemetry, mission, okHandover
        ↪ , keyboardCommand}; -- Defines the possible message
        ↪ types to send from UAV to the GCS.
416 messageFromGcsPartnerToUav: {none, acknowledgeConnectMessage}; --
        ↪ Defines the possible message types to send from UAV to
        ↪  the GCS.
417 varUav_checkDistanceToGoal: 0..99; -- Defines the distance to goal
        ↪ as an integer. Scale/interpretation: 1=1 meter; Range
        ↪ = 0..99 meters.
418 commanderHasSwitched: boolean; -- Defines if the commander has
        ↪ switched.
419 keyboardPress: {none, left, right, up, down}; -- Possible
        ↪ directions for the keyboard presses.
420
421 -- Process Definition
422   -- Keyboard Simulation
423   keyboard : process keyboardInput(keyboardPress);
424   -- UAV Process (UAV State)
425   uav : process UavControl(commanderId, poweredOff,
        ↪ varUav_connectionToGcsCommander,
        ↪ varGcs_isCommanderConnectionAuthorized,
        ↪ messageFromUavToGcsCommander,
        ↪ messageFromUavToGcsPartner,
        ↪ messageFromGcsCommanderToUav,
        ↪ messageFromGcsPartnerToUav, varUav_checkDistanceToGoal
        ↪ , commanderHasSwitched, keyboardPress);
426   -- GCS process (GCS ID, Commander ID, GCS State)
427   gcs1 : process GcsControl(1, commanderId, poweredOff,
        ↪ varUav_connectionToGcsCommander,
        ↪ varGcs_isCommanderConnectionAuthorized,
        ↪ messageFromUavToGcsCommander,
        ↪ messageFromUavToGcsPartner,
        ↪ messageFromGcsCommanderToUav,
        ↪ messageFromGcsPartnerToUav, varUav_checkDistanceToGoal
        ↪ ,varGcs_isPartnerConnected, commanderHasSwitched,
        ↪ keyboardPress);
428   gcs2 : process GcsControl(2, commanderId, poweredOff,
        ↪ varUav_connectionToGcsCommander,
        ↪ varGcs_isCommanderConnectionAuthorized,
        ↪ messageFromUavToGcsCommander,
        ↪ messageFromUavToGcsPartner,
        ↪ messageFromGcsCommanderToUav,
        ↪ messageFromGcsPartnerToUav, varUav_checkDistanceToGoal
        ↪ ,varGcs_isPartnerConnected, commanderHasSwitched,
        ↪ keyboardPress);
429
430 ASSIGN
431 init(commanderId) := 1;
```

```
432 init(varUav_connectionToGcsCommander) := noConnection;
433 init(varGcs_isCommanderConnectionAuthorized) := FALSE;
434 init(messageFromUavToGcsCommander) := none;
435 init(messageFromUavToGcsPartner) := none;
436 init(messageFromGcsCommanderToUav) := none;
437 init(messageFromGcsPartnerToUav) := none;
438 init(varGcs_isPartnerConnected) := FALSE;
439
440 next(varGcs_isPartnerConnected) :=
441   case
442     messageFromUavToGcsPartner = connectHandover : TRUE;
443     TRUE: FALSE;
444   esac;
445
446 next(commanderId) :=
447   case
448     varUav_connectionToGcsCommander = connected &
          ↪ varGcs_isPartnerConnected & commanderId  = 1 & !
          ↪ commanderHasSwitched : 2;
449     varUav_connectionToGcsCommander = connected &
          ↪ varGcs_isPartnerConnected & commanderId  = 2 & !
          ↪ commanderHasSwitched : 1;
450     TRUE: commanderId;
451   esac;
452
453 next(commanderHasSwitched) :=
454   case
455     commanderHasSwitched & !(varGcs_isPartnerConnected) : FALSE;
456     varUav_connectionToGcsCommander = connected &
          ↪ varGcs_isPartnerConnected & commanderId = 1 : TRUE;
457     varUav_connectionToGcsCommander = connected&
          ↪ varGcs_isPartnerConnected & commanderId = 2 : TRUE;
458     TRUE: commanderHasSwitched;
459   esac;
460
461
462 -- REQ-GCS-017 - While the GCS Commander is in "Operating" state
          ↪ and GCS Partner is connected to UAV, the commander
          ↪ shall disconnect and the partner shall be the new
          ↪ commander.
463 LTLSPEC G(varUav_connectionToGcsCommander = connected &
          ↪ varGcs_isPartnerConnected & commanderId  = 1 & !
          ↪ commanderHasSwitched -> next(commanderId = 2))
464 LTLSPEC G(varUav_connectionToGcsCommander = connected &
          ↪ varGcs_isPartnerConnected & commanderId  = 1 & !
          ↪ commanderHasSwitched -> next(commanderId = 1))
```

Appendix B

Appendix B - Iteration 2 NuSMV model

```
MODULE uav (stateUav, stateGcs, isUavInHandoverRegion)
FAIRNESS
    -- Ensure that all processes have fair opportunities to execute
    running;
VAR

ASSIGN

    next(stateUav) := case
        stateUav = requestHandover & stateGcs = authorizeHandover :
          ↪  connectHandover; --REQ-UAV-002: The UAV shall connect
          ↪  to the GCS Partner, if the GCS Commander authorizes
          ↪ the handover to the GCS Partner.
        stateUav = sendTelemetry & isUavInHandoverRegion :
          ↪ requestHandover; -- REQ-UAV-001: The UAV shall request
          ↪  handover to the GCS Partner, when it reaches the
          ↪ handover region.
        stateUav = idle | stateGcs = receiveTelemetry:
          ↪ sendTelemetry;
        stateUav = sendTelemetry | stateGcs = disconnectCommander :
          ↪  idle;
        TRUE : stateUav;
    esac;

MODULE gcs(id, commanderId, stateUav, stateGcs,
          ↪ isUavInHandoverRegion, commanderHasSwitched)

FAIRNESS
    -- Ensure that all processes have fair opportunities to execute
    running;

ASSIGN

    next(stateGcs) :=
     case
```

```
27          stateGcs = droneConnectToPartner & (commanderId = id) & !
              ↪ commanderHasSwitched : disconnectCommander; -- REQ-GCS
              ↪ -002: The GCS Commander shall disconnect from the UAV,
              ↪  transfering the authority over the UAV to the GCS
              ↪ Partner, when the GCS Partner informs the GCS
              ↪ Commander that the UAV has connected to the GCS
              ↪ Partner
28          stateGcs = authorizeHandover & stateUav = connectHandover &
              ↪  !(commanderId = id) : droneConnectToPartner; -- REQ-
              ↪ GCS-004: The GCS Partner shall inform GCS Commander
              ↪ that the UAV has connected to the GCS Partner, when
              ↪ the UAV connects to the GCS Partner.
29          stateGcs = acceptPartnerRequest & (commanderId = id) :
              ↪ authorizeHandover; -- REG-GCS-001: The GCS Commander
              ↪ shall authorize handover, if the GCS Partner informed
              ↪ the GCS Commander that the UAV has requested handover
              ↪ to the GCS Partner.
30          stateGcs = receiveTelemetry & isUavInHandoverRegion & !(
              ↪ commanderId = id) & stateUav = requestHandover:
              ↪ acceptPartnerRequest; -- REQ-GCS-003: The GCS Partner
              ↪ shall inform GCS Commander that the UAV has requested
              ↪ handover when the GCS Partner receives a handover
              ↪ request from the UAV
31          stateGcs = idle & commanderId = id : receiveTelemetry;
32          stateGcs = receiveTelemetry | (stateGcs =
              ↪ disconnectCommander & commanderHasSwitched) : idle;
33          TRUE : stateGcs;
34       esac;
35
36  MODULE main -- Third person perspective.
37
38  FAIRNESS
39      running;
40  VAR
41      commanderId : 1..2;
42      isUavInHandoverRegion : boolean;
43      stateUav : {idle, sendTelemetry, requestHandover,
              ↪ connectHandover};
44      stateGcs : {idle, receiveTelemetry, acceptPartnerRequest,
              ↪ authorizeHandover, droneConnectToPartner,
              ↪ disconnectCommander};
45      commanderHasSwitched : boolean;
46      messageTimeout : boolean;
47
48      -- Define processes
49      uav : process uav(stateUav, stateGcs, isUavInHandoverRegion);
50      gcs1 : process gcs(1,commanderId, stateUav, stateGcs,
              ↪ isUavInHandoverRegion, commanderHasSwitched);
51      gcs2 : process gcs(2,commanderId, stateUav, stateGcs,
              ↪ isUavInHandoverRegion, commanderHasSwitched);
52
53  ASSIGN
54
55      init(commanderId) := 1;
```

```
56      init(isUavInHandoverRegion) := TRUE;
57      init(stateUav) := idle;
58      init(stateGcs) := idle;
59      init(commanderHasSwitched) := FALSE;
60      init(messageTimeout) := FALSE;
61
62      next(isUavInHandoverRegion) :=
63          case
64              TRUE: {TRUE,FALSE};
65              --TRUE: TRUE;
66          esac;
67
68      next(commanderId) :=
69          case
70              commanderId = 1 & stateGcs = disconnectCommander & !
            ↪ commanderHasSwitched : 2;
71              commanderId = 2 & stateGcs = disconnectCommander & !
            ↪ commanderHasSwitched : 1;
72              TRUE: commanderId;
73          esac;
74
75      next(commanderHasSwitched) :=
76          case
77              stateGcs = disconnectCommander : TRUE;
78              TRUE: FALSE;
79          esac;
80
81  -- SPECS
82
83  -- This specification states if the UAV has requested handover and
            ↪ the GCS Partner accepted partner request then it will
            ↪ eventually authorize the handover.
84  LTLSPEC G (stateUav = requestHandover & stateGcs =
            ↪ acceptPartnerRequest -> F stateGcs = authorizeHandover
            ↪ );
85
86  --This specification is used to verify that a commander switch is
            ↪ possible outside of the handover region
87  LTLSPEC   (!isUavInHandoverRegion & commanderId = 1 & stateUav =
            ↪ requestHandover -> !isUavInHandoverRegion U (F ((
            ↪ commanderId = 2) & !isUavInHandoverRegion &
            ↪ commanderHasSwitched & stateGcs = disconnectCommander)
            ↪ )); -- This is the spec that states that is possible
            ↪ switch commmanders, assuming it has reached stateUav =
            ↪  requestHandover.
88
89  -- Assuming the UAV is in handover region and has started the
            ↪ handover process, then eventually, the GCS commander
            ↪ shall be switched.
90  LTLSPEC   (isUavInHandoverRegion & commanderId = 1 & stateUav =
            ↪ requestHandover & stateGcs = receiveTelemetry -> F (
            ↪ commanderId = 2));
91
```

```
92  -- There is always a path where the UAV is in Handover Region,
        ↪ while in idle state, where eventually shall start the
        ↪ handover process.
93  CTLSPEC EG  (isUavInHandoverRegion & stateUav = idle -> EF (
        ↪ stateUav = requestHandover));
94
95  -- Ensure telemetry is sent and handover region detection starts
96  LTLSPEC G (stateUav = idle -> F stateUav = sendTelemetry);
97
98  -- Ensure the system connects to the partner as a client when in
        ↪ the handover region
99  LTLSPEC  (isUavInHandoverRegion & stateUav = requestHandover -> F
        ↪ stateUav = connectHandover);
100
101 CTLSPEC  (!isUavInHandoverRegion & stateUav = idle & stateGcs =
        ↪ idle -> EF commanderHasSwitched);
102
103 -- Verify that the system waits for authorization from the
        ↪ commander when not in the handover region
104 LTLSPEC  (!isUavInHandoverRegion & stateUav = sendTelemetry -> F
        ↪ stateUav = requestHandover & !isUavInHandoverRegion);
105
106 -----------------------------------------------------------
107 -- Ensure the system can change commanders when required --
108 -----------------------------------------------------------
109
110 -- Switch from Commander 1 to 2
111 LTLSPEC G (stateGcs = disconnectCommander & commanderId = 1 & !
        ↪ commanderHasSwitched -> F commanderId = 2);
112 CTLSPEC EG (stateGcs = idle & commanderId = 1 & !
        ↪ commanderHasSwitched -> EF commanderId = 2);
113
114 -- Switch from Commander 2 to 1
115 LTLSPEC G (stateGcs = disconnectCommander & commanderId = 2 & !
        ↪ commanderHasSwitched -> F commanderId = 1);
116
117 -- Switch from Commander 1 to 2 to 1
118 LTLSPEC G (stateGcs = disconnectCommander & commanderId = 1 & !
        ↪ commanderHasSwitched -> F commanderId = 2 -> F(
        ↪ stateGcs = disconnectCommander & commanderId = 2 & !
        ↪ commanderHasSwitched) -> F(commanderId = 1) );
119 CTLSPEC EG (stateGcs = idle & commanderId = 2 & !
        ↪ commanderHasSwitched -> EF commanderId = 1  -> EF
        ↪ commanderId = 2 );
120
121 -----------------------------------------------------------
122 -- Ensure all states are reachable in order for the UAV. --
123 -----------------------------------------------------------
124
125 CTLSPEC EG (stateUav = idle -> EF (stateUav = sendTelemetry));
126
127 -- Possible to reach idle state from sendTelemetry state.
128 CTLSPEC EG (stateUav = idle & isUavInHandoverRegion -> EF(stateUav
        ↪ = idle));
```

```
129
130 -- Possible to reach requestHandover
131 CTLSPEC EG (stateUav = idle -> EF (stateUav = requestHandover));
132
133 -- Ensure all states are reachable in order for the UAV.
134 CTLSPEC EG (stateUav = idle -> EF (stateUav = idle));
135
136 -- Possible to reach idle state from sendTelemetry state.
137 CTLSPEC EG (stateUav = idle -> EF(stateUav = sendTelemetry));
138
139 -- Possible to reach requestHandover
140 CTLSPEC EG (stateUav = idle -> EF (stateUav = connectHandover) ->
        ↪ EF (stateUav = idle));
141
142 -----------------------------------------------------------
143 -- Ensure all states are reachable in order for the GCS. --
144 -----------------------------------------------------------
145
146 -- Possible to reach idle state from receiveTelemetry state.
147 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = receiveTelemetry));
148
149 -- Possible to reach idle state from acceptPartnerRequest state.
150 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = acceptPartnerRequest))
        ↪ ;
151
152 -- Possible to reach idle state from authorizeHandover state.
153 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = authorizeHandover));
154
155 -- Possible to reach idle state from authorizeHandover state.
156 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = droneConnectToPartner)
        ↪ );
157
158 -- Possible to reach idle state from authorizeHandover state.
159 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = disconnectCommander)
        ↪ -> EF (stateGcs = idle));
```

Appendix C

Appendix C - Iteration 3 NuSMV model

```
1  MODULE uav (stateUav, stateGcs, isUavInHandoverRegion,
          ↪ messageTimeout)
2  FAIRNESS
3      -- Ensure that all processes have fair opportunities to execute
4      running;
5  VAR
6
7  ASSIGN
8
9      next(stateUav) := case
10         stateUav = connectPartner & stateGcs = authorizeHandover &
          ↪ (isUavInHandoverRegion & !messageTimeout) :
          ↪ connectHandover;  --REQ-UAV-002: The UAV shall connect
          ↪  to the GCS Partner, if the GCS Commander authorizes
          ↪ the handover to the GCS Partner.
11         stateUav = sendTelemetry & (isUavInHandoverRegion & !
          ↪ messageTimeout) : connectPartner; -- REQ-UAV-001: The
          ↪ UAV shall request handover to the GCS Partner, when it
          ↪  reaches the handover region.
12         stateUav = idle | stateGcs = receiveTelemetry:
          ↪ sendTelemetry;
13         stateUav = sendTelemetry | stateGcs = disconnectCommander |
          ↪  !isUavInHandoverRegion | messageTimeout : idle; --
          ↪ REQ-UAV-003: The UAV shall transition to state idle if
          ↪  a message timeout occurred or if the UAV is not in
          ↪ handover region.
14         TRUE : stateUav;
15     esac;
16
17 MODULE gcs(id, commanderId, stateUav, stateGcs,
          ↪ isUavInHandoverRegion, commanderHasSwitched,
          ↪ messageTimeout)
18
19 FAIRNESS
20     -- Ensure that all processes have fair opportunities to execute
21     running;
22
23 ASSIGN
24
25     next(stateGcs) :=
```

```
26       case
27           stateGcs = droneConnectToPartner & (commanderId = id) & !
             ↪ commanderHasSwitched & (isUavInHandoverRegion & !
             ↪ messageTimeout) : disconnectCommander; -- REQ-GCS-002:
             ↪  The GCS Commander shall disconnect from the UAV,
             ↪ transfering the authority over the UAV to the GCS
             ↪ Partner, when the GCS Partner informs the GCS
             ↪ Commander that the UAV has connected to the GCS
             ↪ Partner
28           stateGcs = authorizeHandover & stateUav = connectHandover &
             ↪  !(commanderId = id) & (isUavInHandoverRegion & !
             ↪ messageTimeout) : droneConnectToPartner; -- REQ-GCS
             ↪ -004: The GCS Partner shall inform GCS Commander that
             ↪ the UAV has connected to the GCS Partner, when the UAV
             ↪  connects to the GCS Partner.
29           stateGcs = acceptPartnerRequest & (commanderId = id) & (
             ↪ isUavInHandoverRegion & !messageTimeout) :
             ↪ authorizeHandover; -- REG-GCS-001: The GCS Commander
             ↪ shall authorize handover, if the GCS Partner informed
             ↪ the GCS Commander that the UAV has requested handover
             ↪ to the GCS Partner.
30           stateGcs = receiveTelemetry & (isUavInHandoverRegion & !
             ↪ messageTimeout) & !(commanderId = id) & stateUav =
             ↪ connectPartner: acceptPartnerRequest;  -- REQ-GCS-003:
             ↪  The GCS Partner shall inform GCS Commander that the
             ↪ UAV has requested handover when the GCS Partner
             ↪ receives a handover request from the UAV
31           stateGcs = idle & commanderId = id : receiveTelemetry;
32           stateGcs = receiveTelemetry | (stateGcs =
             ↪ disconnectCommander & commanderHasSwitched) | !
             ↪ isUavInHandoverRegion | messageTimeout : idle; -- REQ-
             ↪ GCS-005: The GCS shall transition to state idle if a
             ↪ message timeout occurred or if the UAV is not in
             ↪ handover region.
33
34           TRUE : stateGcs;
35       esac;
36
37 MODULE main -- Third person perspective.
38
39 FAIRNESS
40     running;
41 VAR
42     commanderId : 1..2;
43     isUavInHandoverRegion : boolean;
44     stateUav : {idle, sendTelemetry, connectPartner,
             ↪ requestAuthorizationFromCommander, connectHandover};
45     stateGcs : {idle, receiveTelemetry, acceptPartnerRequest,
             ↪ authorizeHandover, droneConnectToPartner,
             ↪ disconnectCommander};
46     commanderHasSwitched : boolean;
47     messageTimeout : boolean;
48
49     -- Process Definition
```

```
50      uav : process uav(stateUav, stateGcs, isUavInHandoverRegion,
          ↪ messageTimeout);
51      gcs1 : process gcs(1,commanderId, stateUav, stateGcs,
          ↪ isUavInHandoverRegion, commanderHasSwitched,
          ↪ messageTimeout);
52      gcs2 : process gcs(2,commanderId, stateUav, stateGcs,
          ↪ isUavInHandoverRegion, commanderHasSwitched,
          ↪ messageTimeout);
53
54  ASSIGN
55
56      init(commanderId) := 1;
57      init(isUavInHandoverRegion) := TRUE;
58      init(stateUav) := idle;
59      init(stateGcs) := idle;
60      init(commanderHasSwitched) := FALSE;
61      init(messageTimeout) := FALSE;
62
63
64      next(messageTimeout) :=
65          case
66              TRUE: {TRUE,FALSE};
67              --TRUE: TRUE;
68          esac;
69
70      next(isUavInHandoverRegion) :=
71          case
72              TRUE: {TRUE,FALSE};
73              --TRUE: TRUE;
74          esac;
75
76      next(commanderId) :=
77          case
78              commanderId = 1 & stateGcs = disconnectCommander & !
          ↪ commanderHasSwitched : 2;
79              commanderId = 2 & stateGcs = disconnectCommander & !
          ↪ commanderHasSwitched : 1;
80              TRUE: commanderId;
81          esac;
82
83      next(commanderHasSwitched) :=
84          case
85              stateGcs = disconnectCommander : TRUE;
86              TRUE: FALSE;
87          esac;
88
89  -- SPECS
90
91  -- This specification is used to verify when the state UAV is
          ↪ connecting to partner and the partner GCS accepts the
          ↪ partner request, eventually the handover shall be
          ↪ authorized.
```

```
92  -- LTLSPEC G (stateUav = connectPartner & stateGcs =
        ↪ acceptPartnerRequest -> F stateGcs = authorizeHandover
        ↪ );
93  -- After implementing the cancellation of the handover procedure
        ↪ and returning the Gcs state to idle, it fails.
94
95  -- This specification is used to verify when the state UAV is
        ↪ connecting to partner and the partner GCS accepts the
        ↪ partner request while the UAV is in the handover
        ↪ region, eventually the handover shall be authorized.
96  LTLSPEC (stateUav = connectPartner & stateGcs =
        ↪ acceptPartnerRequest & isUavInHandoverRegion -> F
        ↪ stateGcs = authorizeHandover);
97
98
99  --This specification is used to verify that a commander switch is
        ↪ possible outside of the handover region
100 LTLSPEC  (!isUavInHandoverRegion & commanderId = 1 & stateUav =
        ↪ connectPartner -> !isUavInHandoverRegion U (F ((
        ↪ commanderId = 2) & !isUavInHandoverRegion &
        ↪ commanderHasSwitched & stateGcs = disconnectCommander)
        ↪ )); -- This is the spec that states that is possible
        ↪ switch commmanders, assuming it has reached stateUav =
        ↪  connectPartner.
101
102 -- Assuming the UAV is in handover region and has started the
        ↪ handover process, then eventually, the GCS commander
        ↪ shall be switched.
103 LTLSPEC  (isUavInHandoverRegion & commanderId = 1 & stateUav =
        ↪ connectPartner & stateGcs = receiveTelemetry -> F (
        ↪ commanderId = 2));
104
105 -- There is always a path where the UAV is in Handover Region,
        ↪ while in idle state, where eventually shall start the
        ↪ handover process.
106 CTLSPEC EG  (isUavInHandoverRegion & stateUav = idle -> EF (
        ↪ stateUav = connectPartner));
107
108 -- Ensure telemetry is sent and handover region detection starts
109 LTLSPEC G (stateUav = idle -> F stateUav = sendTelemetry);
110
111 -- Ensure the system connects to the partner as a client when in
        ↪ the handover region
112 LTLSPEC  (isUavInHandoverRegion & stateUav = connectPartner -> F
        ↪ stateUav = connectHandover);
113
114 CTLSPEC  (!isUavInHandoverRegion & stateUav = idle & stateGcs =
        ↪ idle -> EF commanderHasSwitched);
115
116 -- Verify that the system waits for authorization from the
        ↪ commander when not in the handover region
117 LTLSPEC  (!isUavInHandoverRegion & stateUav = sendTelemetry -> F
        ↪ stateUav = requestAuthorizationFromCommander);
118
```

```
119 -- Verify that the system waits for a connect message from the
        ↪ commander when not in the handover region
120 LTLSPEC  (!isUavInHandoverRegion & stateUav = sendTelemetry &
        ↪ stateGcs = receiveTelemetry & !commanderHasSwitched ->
        ↪  F (stateUav = requestAuthorizationFromCommander & !
        ↪ isUavInHandoverRegion));
121
122
123 ------------------------------------------------------------
124 -- Ensure the system can change commanders when required --
125 ------------------------------------------------------------
126
127 -- Switch from Commander 1 to 2
128 --LTLSPEC G (stateGcs = disconnectCommander & commanderId = 1 & !
        ↪ commanderHasSwitched -> F commanderId = 2);
129 -- After implementing the cancellation of the handover procedure
        ↪ and returning the Gcs state to idle, it fails.
        ↪ Replaced with the following:
130 LTLSPEC G (stateGcs = disconnectCommander & commanderId = 1 & !
        ↪ commanderHasSwitched & isUavInHandoverRegion -> F
        ↪ commanderId = 2);
131 CTLSPEC EG (stateGcs = idle & commanderId = 1 & !
        ↪ commanderHasSwitched -> EF commanderId = 2);
132
133 -- Switch from Commander 2 to 1
134 --LTLSPEC G (stateGcs = disconnectCommander & commanderId = 2 & !
        ↪ commanderHasSwitched -> F commanderId = 1);
135 -- After implementing the cancellation of the handover procedure
        ↪ and returning the Gcs state to idle, it fails.
        ↪ Replaced with the following:
136 LTLSPEC G (stateGcs = disconnectCommander & commanderId = 2 & !
        ↪ commanderHasSwitched & isUavInHandoverRegion -> F
        ↪ commanderId = 1);
137
138 -- Switch from Commander 1 to 2 to 1
139 LTLSPEC G (stateGcs = disconnectCommander & commanderId = 1 & !
        ↪ commanderHasSwitched -> F commanderId = 2 -> F(
        ↪ stateGcs = disconnectCommander & commanderId = 2 & !
        ↪ commanderHasSwitched) -> F(commanderId = 1) );
140 CTLSPEC EG (stateGcs = idle & commanderId = 2 & !
        ↪ commanderHasSwitched -> EF commanderId = 1  -> EF
        ↪ commanderId = 2 );
141
142 ------------------------------------------------------------
143 -- Ensure all states are reachable in order for the UAV. --
144 ------------------------------------------------------------
145
146 CTLSPEC EG (stateUav = idle -> EF (stateUav = sendTelemetry));
147
148 -- Possible to reach idle state from sendTelemetry state.
149 CTLSPEC EG (stateUav = idle & isUavInHandoverRegion -> EF(stateUav
        ↪ = idle));
150
151 -- Possible to reach connectPartner
```

```
152 CTLSPEC EG (stateUav = idle -> EF (stateUav = connectPartner));
153
154 -- Ensure all states are reachable in order for the UAV.
155 CTLSPEC EG (stateUav = idle -> EF (stateUav = idle));
156
157 -- Possible to reach idle state from sendTelemetry state.
158 CTLSPEC EG (stateUav = idle -> EF(stateUav = sendTelemetry));
159
160 -- Possible to reach connectPartner
161 CTLSPEC EG (stateUav = idle -> EF (stateUav = connectHandover) ->
        ↪ EF (stateUav = idle));
162
163 ------------------------------------------------------------
164 -- Ensure all states are reachable in order for the GCS. --
165 ------------------------------------------------------------
166
167 -- Possible to reach idle state from receiveTelemetry state.
168 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = receiveTelemetry));
169
170 -- Possible to reach idle state from acceptPartnerRequest state.
171 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = acceptPartnerRequest))
        ↪ ;
172
173 -- Possible to reach idle state from authorizeHandover state.
174 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = authorizeHandover));
175
176 -- Possible to reach idle state from authorizeHandover state.
177 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = droneConnectToPartner)
        ↪ );
178
179 -- Possible to reach idle state from authorizeHandover state.
180 CTLSPEC EG (stateGcs = idle -> EF(stateGcs = disconnectCommander)
        ↪ -> EF (stateGcs = idle));
```

Bibliography

Authors, Spin (2020). *Spin: Software Verification Tool*. https://spinroot.com/. Spin is a widely used open-source software verification tool for formal verification of multi-threaded software applications. Developed at Bell Labs in the Unix group of the Computing Sciences Research Center starting in 1980. Available freely since 1991. Received the ACM System Software Award in April 2002. The latest version is 6.5.1 (July 2020).

Baier, Christel and Joost-Pieter Katoen (2008). *Principles of Model Checking*. MIT Press. isbn: 9780262026499.

Elkholy, Warda et al. (2020). "Model checking intelligent avionics systems for test cases generation using multi-agent systems". In: *Expert Systems with Applications* 156, p. 113458. issn: 0957-4174. doi: https://doi.org/10.1016/j.eswa.2020.113458. url: https://www.sciencedirect.com/science/article/pii/S0957417420302827.

Ferreira, Gleizielly Alves (Oct. 2023). "Authority Handover Procedure and Safety Decision Strategy in Unmanned Aerial Vehicles". English and Portuguese. PhD thesis. Instituto Superior de Engenharia do Porto, Departamento de Engenharia Eletrónica.

Hu, A.J. et al. (1999). "Model-Checking a Secure Group Communication Protocol: A Case Study". In: *Formal Methods for Protocol Engineering and Distributed Systems*. Ed. by J. Wu, S.T. Chanson, and Q. Gao. Vol. 28. IFIP Advances in Information and Communication Technology. Boston, MA: Springer, pp. 199–213. doi: 10.1007/978-0-387-35578-8_27.

Kwiatkowska, M., G. Norman, and D. Parker (2011). "PRISM 4.0: Verification of Probabilistic Real-time Systems". In: *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. LNCS. Springer, pp. 585–591.

Kwiatkowska, Marta, Gethin Norman, and David Parker (Apr. 2017). "Probabilistic Model Checking:: Advances and Applications". English. In: *Formal System Verification*. Ed. by R. Drechsler. Springer. isbn: 9783319576855.

– (2022). "Probabilistic Model Checking and Autonomy". In: *Annual Review of Control, Robotics, and Autonomous Systems* 5.1, pp. 385–410. doi: 10.1146/annurev-control-042820-010947. eprint: https://doi.org/10.1146/annurev-control-042820-010947. url: https://doi.org/10.1146/annurev-control-042820-010947.

Musuvathi, Madanlal and Dawson Engler (Mar. 2004). "Model Checking Large Network Protocol Implementations". In.

Musuvathi, Madanlal, David Y.W. Park, et al. (Dec. 2002). "CMC: A Pragmatic Approach to Model Checking Real Code". In: *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*. Boston, MA: USENIX Association. url: https://www.usenix.org/conference/osdi-02/cmc-pragmatic-approach-model-checking-real-code.

NuSMV: A Symbolic Model Checker (n.d.). https://nusmv.fbk.eu/. Developed as a joint project between the Embedded Systems Unit in the Digital Industry Center at FBK-IRST, the Model Checking group at Carnegie Mellon University, the Mechanized Reasoning Group at University of Genova, and the Mechanized Reasoning Group at University of Trento.

Pereira, David and Alexandre Bragança (2022). *Introduction to Model Checking*. (ISEP) Instituto Superior de Engenharia do Porto.

Pettersson, Paul and Kim G. Larsen (2000). "Bulletin of the European Association for Theoretical Computer Science". In: 70, pp. 40–44.

Rome, Sapienza University of (2012). *CMurphi: A Verification Tool based on Murphi*. `https://github.com/melver/cmurphi`.

Wang, Xi et al. (2021). "A Method for UAV Monitoring Road Conditions in Dangerous Environment". In: *Journal of Physics: Conference Series*. The International Conference on Communications, Information System and Software Engineering (CISSE 2020), 18-20 December 2020, Guangzhou, China 1792. Published under licence by IOP Publishing Ltd, p. 012050. doi: `10.1088/1742-6596/1792/1/012050`.