TECHNISCHE UNIVERSITÄT
CHEMNITZ

"Design of a Test Generation Methodology for ARTIS using Model-Checking with a Generic Modelling Approach"

Master Thesis

"Final Report"

Fakultät Informatik

Masters in Automotive Software Engineering

At the

Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR)
German Aerospace Center, Institute of Flight Systems
Braunschweig, Germany

Author:

Ganesh Kamalakar Vernekar
ganeshkv01@gmail.com

Mat Nr. 325701

Supervisor at DLR:

Dipl.-Inform. Christoph Torens
Christoph.Torens@dlr.de

Supervisors at TU Chemnitz:

Prof. Dr. Wolfram Hardt
hardt@cs.tu-chemnitz.de
Dr. Ariane Heller
ariane.heller@cs.tu-chemnitz.de

December 14, 2015

Acknowledgment

This thesis work was carried out at Deutsches Zentrum für Luft-und Raumfahrt e.V. (DLR) a.k.a. German Aerospace Center, Institute of Flight Systems in Braunschweig, Germany. This work represents the completion of my Master Studies in Automotive Software Engineering under Fakultät für Informatik from Technische Universität Chemnitz. Additionally prior to my thesis, I have also carried my internship at DLR on the topic "Designing a Formal Model for Generic GNC (Guidance, Navigation, and Control) Software of UAV", which gave me immense knowledge and background for my thesis work.
Foremost, I would like to express my sincere gratitude to my advisor Dipl.-Inform. Christoph Torens (DLR Institute of Flight Systems Braunschweig, Germany), Prof. Dr. Wolfram Hardt (Fakultät Für Informatik, Technische Universität Chemnitz, Germany), Dr. Ariane Heller (Fakultät Für Informatik, Technische Universität Chemnitz, Germany), for the continuous support of my Master Thesis research, for their patience, motivation, enthusiasm, and immense knowledge.
I would also like to thank Florian-Michael Adolf (DLR Institute of Flight Systems Braunschweig, Germany) for his encouragement, insightful comments, and questions that helped me to gain a larger picture of my work.

I would like to specially thank Dipl.-Inform. Christoph Torens for his guidance and for offering me the internship and thesis opportunity. His experience in the field of Software Engineering has helped me a lot during work. His knowledge has helped me during the entire time of research and his motivation in using LaTeXhas made me organize my research content more professionally and allowed for better structuring of my thesis writeup. I could not have imagined having a better advisor and mentor for my Master Thesis. I am very grateful to my University for providing me an opportunity to have an industrial exposure during my Master Studies. It would never be possible without the guidance of Prof. Dr. Wolfram Hardt, Dr. Ariane Heller.

Last but not the least, I would like to thank my parents for supporting me in every aspect throughout my life.

Abstract

In the recent trends, automated systems are increasingly seen to be embedded in human life with the increase of human dependence on software to perform safety-critical tasks like airbag deployment in automobiles to real-time mission planning in UAVs (Unmanned Aircraft Vehicles). The safety-critical nature of the aerospace domain demands for a software without any errors to perform these tasks. Therefore the field of computer science needs to address these challenges by providing necessary formalisms, techniques, and tools that will ensure the correctness of systems despite their complexity. DO-178C/EC-12C is a standard that governs the certification of software for airborne systems in commercial aircraft. The additional supplement DO-333 enables us to use the formal methods in our technique of verifying the autonomous behaviour of UAV's.

The Mission Manager system is primarily responsible for the execution of behaviour sequence in online and offline mission planning of UAV. This work presents the process of software verification by making use of formal modelling using model checking of the Mission Manager component of ARTIS (Autonomous Rotorcraft Testbed for Intelligent Systems) UAV by gaining advantages from a generic modelling approach. The main idea is to make use of the designed generic models into specific cases like ARTIS in our case. The generic models are designed using the ALFU(R)S (Autonomy Levels For Unmanned Rotorcraft System) framework that delineates the commonalities of several UAVs considered around the world which also includes the ARTIS UAV.

Furthermore this work walks through every process involved in model checking like requirements extraction and documentation using a template based method, requirements specification using the temporal logics like LTL and CTL, developing a formal model using NuSMV as a model checking tool to analyze the requirements against the model for the Mission Manager component of MiPlEx (Mission Planning and Execution). Additionally as a validation approach, test sequences are generated by using trap properties or negation properties. This aids for a test generation approach by harnessing counterexample generating capabilities of the NuSMV Model Checker.

Keywords: Formal Specification, Model Checking, Formal Verification, Validation and Verification, UAV, NuSMV, Test Case Generation, Counterexample.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

UAV's are gaining a lot of attention in the recent past because of their small-size and ready to go anywhere capability. Most of them find their application in civil operations such as fire monitoring, search and rescue operations and safe delivery of cargo to designated places. It can also be used in tactical purposes like for example military purposes such as to carry a bombarding material to a remote target location or for surveillance in places where humans are less suited. These UAV's achieve tasks with much less risk to human life and do it more efficiently and effortlessly.

The DLR (Deutsches Zentrum für Luft- und Raumfahrt) Institute of Flight Systems [3], Braunschweig is developing the experimental flying platform named ARTIS (Autonomous Rotorcraft Testbed for Intelligent Systems) rotorcraft as shown in Figure 1.1.

Figure 1.1: The Unmanned ARTIS Helicopters [www.dlr.de]

The increase in autonomous behaviour of the UAV also increases the dependency on the software. This implies that the software should be highly reliable and without any errors or deadlocks. To make sure that the developed software is out of errors we need a rigorous error detection and correction methods in the software and more importantly the coverage criteria is important as we need to find the presence of all the errors within the software.

Reviewing and testing are well known error detection methods and are the major techniques of software verification in practice. A review constitutes of software inspection by a group of software engineers. It is a complete manual process which is effective but not sufficient process because subtle errors such concurrency and algorithm defects are not easy to detect in review process. According to Baier in [5], 30% to 50% of the total software development costs are spent on testing. On the other hand testing is a dynamic process as compared to review which is solely manual process. In testing, a code is executed or a software is run. This helps us to know the correctness of the code as we impose the software to traverse a set of execution paths during the run of the software. But even testing is not complete because the coverage of all the execution path is not feasible in large application software. That is to say that testing can only show the presence of errors but it does not show the absence of errors. According to Baier in [5], the cost of fixing a software during the maintenance is approximately 500 times more that the early design phase. As shown in the Figure 1.2.

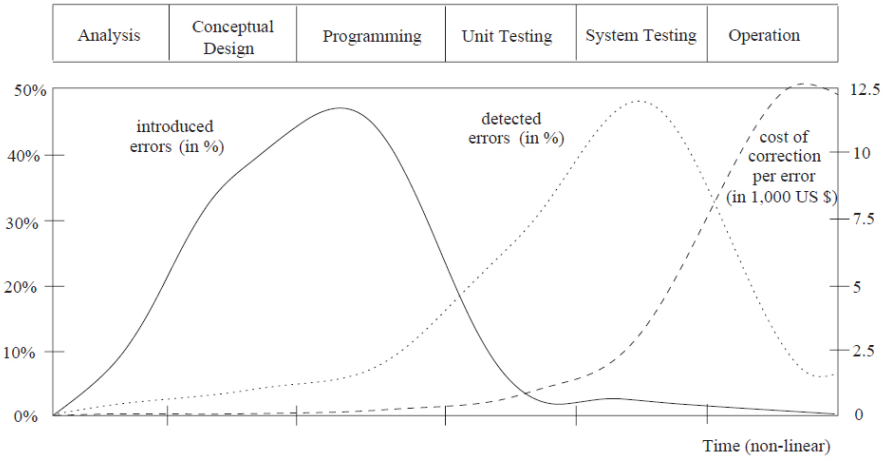

Figure 1.2: Introduced, detected errors and repair costs during Software Life Cycle [5]

1.2 Problem Statement

The DLR Institute of Flight Systems in Braunschweig, Germany performs research with the UAS (Unmanned Aircraft System) to increase the autonomous behaviour of dull and dangerous tasks. This requires that the software is free from any defects like dead locks, assertion violations, integer overflow, division by zero etc. One of the focus at DLR is to design safe systems.

The larger goal, out of the scope of this thesis work would be to investigate the use of formal methods in certification of airborne systems. Currently, DO-178C/ ED-12C is used as a standard for software development and testing of safety-critical systems for aerospace domains. But the test approach in these standards is stochastic in nature as mentioned in [8] by Torens. This work also mentions that a large amount of effort is diverted in creating large set of test cases in identifying these errors. Hence this work although indirectly, points towards reducing the efforts to detect these errors using formal methods. As DO-178C enables the use of formal methods from one of its additional supplements DO-333.

The major research question within the scope of this work would be to use generic model of UAV and make use of it for designing specific cases like ARTIS. As we could significantly gain from these generic models in designing other specific cases. This would significantly reduce the work in verifying other software systems using model checking. But the question remains up to what extent can we make use of these generic models? This is depicted in the Chapter 6. It also explains some of the constraints encountered during this approach. The generality of these generic models remains within the scope of ALFU(R)S framework which includes ARTIS fleet of UAVs.

The idea of having a requirement based test generation framework also needs us to answer if we could generate test cases from these generic models. A test generation approach is explained in Chapter 8. A test generation framework would be significantly important to work with other specific cases. We could take the advantage of formal methods using model checking and verify the requirements against models.

1.3 Goals

The objective of this work is to research the use of generic formal models for modelling, model checking and test generation. The main idea is to use the well known formal methods in verifying the MiPlEx software using the generic modelling approach. The ARTIS consists of fleet of UAVs under its flagship like Super ARTIS, Mini ARTIS, Midi ARTIS etc. The use of generic modelling helps us in reuse of the generic framework in specific cases which reduces considerable effort of designing new framework for every specific cases.

The main focus would be towards the flight guidance software. Firstly, the design

of generic formal model with specialization of a MiPlEx component that can be used for model checking and test generation approach. Secondly, design of test generation methodology and constraints for specialization of generic formal methods. Furthermore, a formal model should be created that is suitable for model-checking and test generation. As a modelling language NuSMV is used because of its counterexample generation capabilities which provides bridging towards test generation framework.

1.4 Document Structure

The thesis work is organised as follows: Chapter 1 explains the basic introduction to the topic with the motivation to the work and research goals. Chapter 2 describes the basic fundamentals required to understand the technical details of the work including the concepts of formal methods. The comparison between different formal techniques is described in Chapter 2. Also a introduction to the model-checking tool (NuSMV) is presented. Chapter 3 describes the related work i.e. the research work carried out in this direction of generic model checking and test generation. Chapter 4 gives the detailed information about the generic and specific UAV's. This chapter provides the information about the system on which the research was carried out.

Chapter 5 gives a brief summary of contribution to thesis is given to present the work done under this thesis. Chapter 6 presents the approach towards the generic modelling. It maps the information flow model of GNC component of UAV to the generic agent model. Chapter 7 explains the process of model-checking of specific UAV guidance system. It explains the process of requirement extraction, formalisation, different graphical models generated during this work. Chapter 8 explains the process of test sequence generation by considering a mission scenario. Finally Chapters9 and 10 depicts the experimental results, conclusion along with future scope of this work.

1.5 Summary

UAV's find enormous application in civil and military domains because of there ready to go anywhere capability. Civil UAV's can be used in fire monitoring or first aid cargo delivery and in many other emergency situations. But as these need to operate between narrow building, trees, poles with high precision. For such a precision in operation the software needs to be out of errors.

The process of review and testing are well known methods in software verification and are popularly used inn finding errors but for a safety critical system a more rigorous methodology needs to be used to find all the errors, deadlocks present in the software. Hence, formal methods can be used for software verification because of their high precision and coverage criteria.

Chapter 2

Fundamentals

This chapter explains the basic concepts required for understanding this work. Firstly it provides the process of requirement elicitation. Requirements are necessary to understand the property or behaviour of a software system. These requirements are necessary linguistic approach to defines the requirements of a system. Secondly, these requirements are transformed into mathematical form i.e. in the computer understandable language using the Temporal Logics (TL). The temporal logic consists of LTL (Linear Temporal Logics) and CTL (Computation Tree Logic) logics.

It explains the concept of formal methods as the process of model-checking (used in this work) is one of a formal method in verifying software systems. This chapter also states some of the important definition useful in understanding formal artefacts. Some of the limitations of formal techniques along with its benefits are described. It compares various formal techniques used in software verification. Followed by the model checking using NuSMV model checker.

2.1 Requirement Elicitation

Requirements of a software system define the basic desired property or behaviour of a system. These requirements could be collected from customers base but during the perpetuation of requirements for example from customer to project leader to analyst to programmer, it could become ambiguous as the requirements specified by the different stakeholders do not certainly remain to be the same. This could also create confusion amongst understanding the right meaning of the actual requirement [35]. This could lead to erroneous process of understanding and implementation of the requirements of the system. A structured approach is required in defining these requirements. Several methodologies are available in defining these requirements like analytical approach, natural language approach and the template based approach. The template based approach is better amongst all the requirement methodologies. The natural language

method basically provides the better approach than compared to the analytical approach. The reasons that the requirements are clearer basically due to investigation on the linguistic effects are included and the knowledge gained is being integrated into the statements. The disadvantage of natural language approach is that this approach cannot provide a common and uniform understanding of the system. The template based requirements is as presented below as it provides a structured and uniform approach to define these requirements using pre-defined templates.

Template Based Requirements:
It is necessary to frame good requirements to avoid typical errors. A simple template based method [35] can solve the above problem that is based on linguistic and philosophical fundamentals. This approach avoids the phrasing error right from the start. Due to its benefits of phrasing ambiguities and structured approach, it was also used during the internship work in the process of specifying requirements. The detailed template based requirement formation is explained below in 6 steps:

Step 1: Determine the process: The desired functionality (like print, save, calculate and so on) is most important and shall be framed as a process.

Step 2: Determine the activity of system: "The System" or a subsystem shall always be rendered as the subject of the requirement. A requirement could be classified into three categories according to paper [35].

- Independent System Activity: The system executes the process independently

  THE SYSTEM ..... <process word>

- User Interaction: The system provides the user with the ability to use the process functionality.

  THE SYSTEM ..... PROVIDE <whom?> WITH THE ABILITY TO <process word>

- Interface Requirement: The system executes a process dependent upon a third party (such as an external system).

  THE SYSTEM ..... BE ABLE TO <process word>

Depending on the system activity we can choose one of the three-requirement templates as shown in Figure 2.1.

Step 3: Determining legal obligation: We should distinguish between the legal relevance of a requirement such as legal binding, strongly recommended and future requirement. This could be done by using modal verbs like *shall, should* and *will*. This helps us to easily determine the degree of legal obligation of your requirement as shown in figure 2.2.

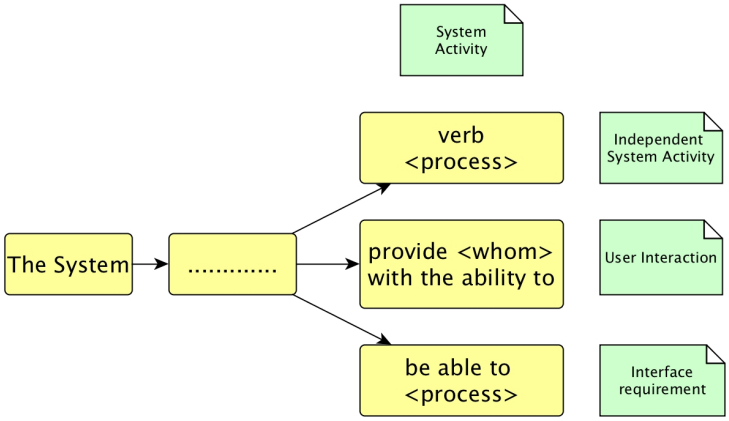

Figure 2.1: Requirement indicating activity [35]

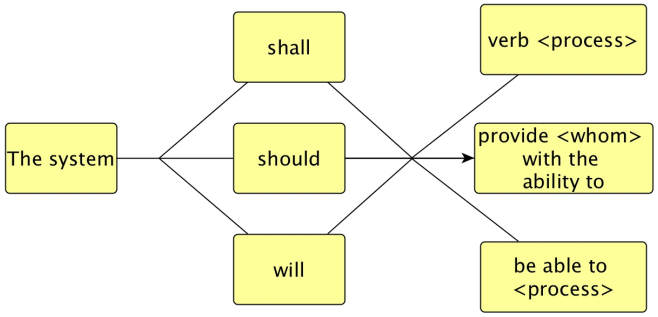

Figure 2.2: Requirement indicating legal relevance [35]

Summarising step 1 to 4 an example can be formed:
Requirement No 1, Version 1: The system SHALL provide THE RECEPTIONIST with the ability to PRINT.

Step 4: Fine Tuning: From the above example it is still not clear as to "WHAT" has to be printed or to "WHERE" it is to be printed. Thus it is clearly seen that more information is needed about the requirement. It is shown in Figure 2.3.
Requirement No. 1, version 2: The system SHALL provide THE RECEPTIONIST with the ability to PRINT A BILL ON THE NETWORK PRINTER.

Step 5: Phrasing of logical and temporal Conditions: There is a need to specify the logical and temporal conditions (Figure 2.4) in order to clearly differentiate certain conditions. "WHEN" for temporal conditions and the conditional conjunction "IF" for

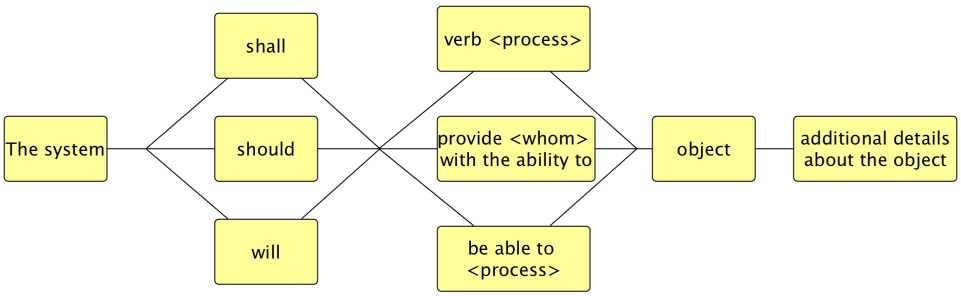

Figure 2.3: Requirement without conditions [35]

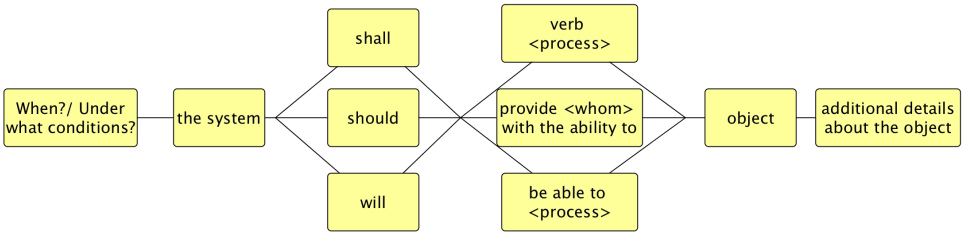

Figure 2.4: Requirement with conditions [35]

logical conditions. Prerequisites need to be determined, which need to be met before the requirement is to be valid.

Requirement No. 1, version 3: If the option "BILL REQUIRED" HAS BEEN SELECTED ON THE MOBILE DEVICE, the system shall provide the receptionist with the ability to print a bill on the network printer.

Step 6: SOPHIST-Rulebook There are still some exceptions for the requirement to be complete. The SOPHIST-Rulebook [11] is used in order to ensure completeness of the semantic meaning and to avoid linguistic defects. On the whole step 6 defines the auditing of the requirement.

2.2 Formalising Requirements using Temporal Logics

The requirements are formalised using temporal logics. Temporal logics (TL) [5] are formalism for specifying and verifying properties of reactive systems. The requirements template that we have learnt above can be transformed into a special syntax. By using these sequences we can check if the requirements are holding true to our

model or not, by this we can verify if the model satisfies the property or not. The specification can be represented using logical framework. The logic becomes the necessity part to develop language that can model the situations and that makes the way to reason about them formally. The propositional logic can express the specification such a way that the logical structure can be brought out. The propositional logic is based on the propositions that express either being "TRUE" or "FALSE"The propositional logic logic can not express assertions of all types. The predicate logic came up as a powerful logic than compared to the propositional logic. Predicate logic is the generalisation of the propositional variable. The vital part of both these two logics namely predicate logic and propositional logic represent the specifications whose truth value is constant in time[5]. The temporal logics delineate the temporal relations between the events occurring over time. The temporal logic notation is often simpler and clearer. Model-checking is one of the methods to verify the temporal properties of the system. Temporal logic allows for formal specification of properties such as safety (nothing bad will happen), liveness (something good will happen) and fairness (independent processes will progress).

Temporal logics may differ according to how they handle branching in the underlying computation tree. In a linear temporal logic, operators are provided for describing events along a single computation path. In a branching-time logic the temporal operators quantify over the paths that are possible from a given state. LTL (Linear Temporal Logic) [11] and [5] can have infinite sequences of states where each point in time has a unique successor based on a linear-time perspective. LTL is built up from a finite set of automatic propositions. CTL (Computation Tree Logic) [11] and [5] describes properties of a computation tree and formulas can reason about many executions at once. It belongs to the family of branching-time logics and semantics defined in terms of states.

2.2.1 Linear Temporal Logics

LTL characterises every linear path induced by the Finite State Machine (FSM) in linear-time approach. Linear-time properties specify the traces that a transition system should exhibit [5]. The linear temporal logic describes the events only along the single computation path.

Symbol	**Meaning**
&	Logical AND
|	Logical OR
!	Logical NOT

Table 2.1: Logical Operators [5]

Symbol	Meaning
X	Refers to next state
G	Globally
F	Future
U	Until
Y	Previous state
S	Since
p U q	p Until q

Table 2.2: Temporal Operators [5]

The temporal operators such as X, G, F and U basically represent the future and the operators such as Y and S represents the past. A “F p” read as future, states that a certain condition p holds in one of the future time instants. “G p” read as globally, states that a certain condition p holds in all future time instants. “p U q” read as p Until q, states that condition p holds until a state is reached where condition q holds. “X p” read as next p, states that condition p is true n the next state[24].

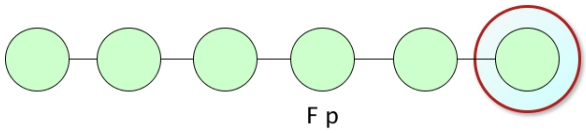

Figure 2.5: Temporal Operator ‘Final’

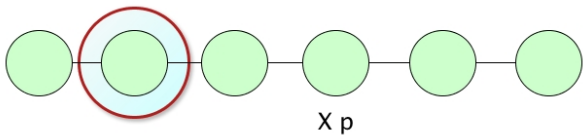

Figure 2.6: Temporal Operator ‘Next’

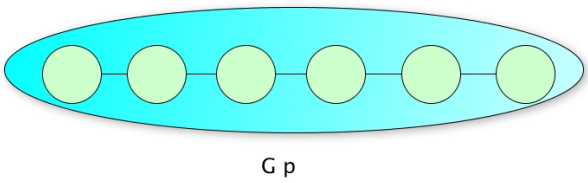

Figure 2.7: Temporal Operator ‘Globally’

A similar approach was used during the internship work. One of the example of

LTL as seen from requirements in this work is presented below:

'Always the Sequence Controller shall provide the ability to the Manual Pilot to switch between Manual Pilot Mode and Mission Mode'
LTL specification:
LTLSPEC G ((SC_State = MissionControllerOff) -> F (SC_State = MissionPlanning));

2.2.2 Computation Tree Logics

Computation tree logic is also called as branching time logic and here the temporal operators quantify over the paths that are possible from a given state. Its formulas allow for specifying properties that considers the non-deterministic, branching evolution of a FSM. The branching time logic represents the time as tree rooted present instance of time and branching out into the future. The branching has many future possibilities and it depends on the system behaviour. The evolution of a FSM from a given state can be described as an infinite tree, where the nodes are the states of the FSM. The paths in the tree that start in a given state are the possible alternative evolutions of the FSM from that state [24]. In CTL one can express properties that should hold for all the paths that start in a state, as well as for properties that should hold just for some of the paths. The Figure 2.8 illustrates the branching progress of time. CTL quantifies the statements over all the paths and can also quantify the single path from a state and this can be further understood easily from the Figures 2.9, 2.10, 2.11, 2.12, 2.13.

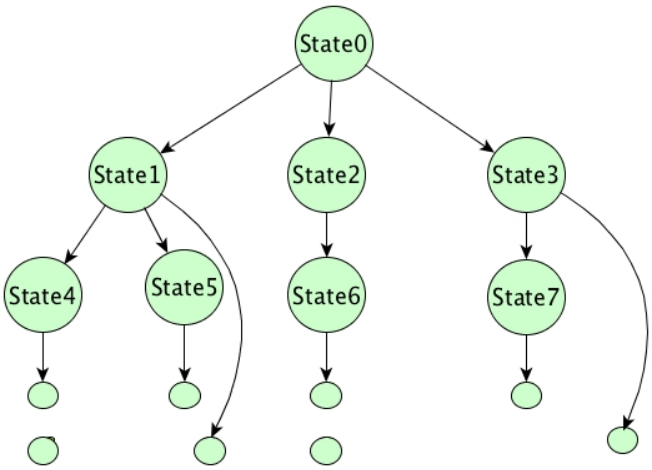

Figure 2.8: Branching Progress of Time

Syntax of CTL: Computational tree logic comprises of atomic propositions, path quantifiers, logic operators and also temporal operators. Atomic propositions such as p,q.

Path quantifiers A, E:

- A: all paths starting from the given state.
- E: there shall exist at least one path from a given state

Logic operators such as ˆ , _:

- ˆ :AND
- _:NOT

Temporal operators such as X, F, G, U:

- X (Next): It refers to the next states of current state.
- F (Future): any one of the future states from the current state.
- G (Global): all future states from the current state.
- U (Until): Some CTL formula holds until another CTL formula from the current state.

X, F, G are the unary operators and the U is the binary operator. Computational tree logic basically combines the temporal operators with the path quantifiers over runs. CTL can be better viewed through the pictorial representation of an example. Lets consider an example with two states black and red. In the figures 2.9, 2.10, 2.11, 2.12, 2.13 the topmost state satisfies the given formula if the black state satisfy p and the red state satisfy q. A similar approach was used during the internship work. One of the example of CTL as seen from requirements in this work is presented below:

'The system shall have different levels of autonomy'
CTL specification:
SPEC AG EF(SC_State = MissionControllerOff);

2.3 Concept of Formal Methods

The application of Formal Methods (FM) techniques could be used in specification and verification of products from development cycle like requirements, high-level and low-level design, and implementation. Another advantage would be that the FM techniques could be used for strict traceability between system descriptions across different life cycle phases. FM are based on mathematical modelling and formal logic that are used to specify and verify requirements and software. Formal methods involve computer-assisted proofs of key properties that explains the behaviour of the system

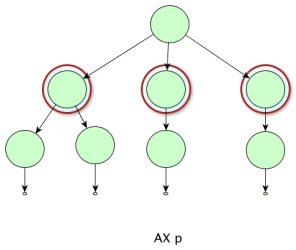

Figure 2.9: CTL Operator ‘AX’

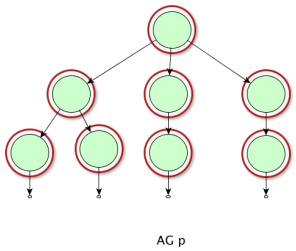

Figure 2.10: CTL Operator ‘AG’

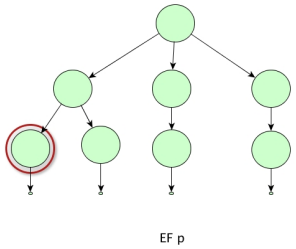

Figure 2.11: CTL Operator ‘EF’

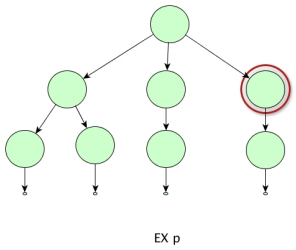

Figure 2.12: CTL Operator ‘EX’

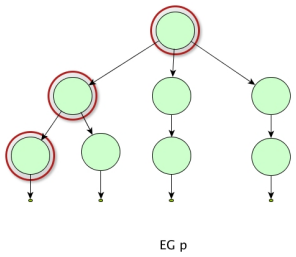

Figure 2.13: CTL Operator 'EG'

and it proves to be one of the essential tools for activities including certification, reuse, and assurance [23]. According to [12], a report written by FAA and NASA concludes that:

"Formal methods should be a part of the education of every computer scientist and software engineers, just as the appropriate branch of applied math's is a necessary part of the education of all other engineers."

This signifies the prominence of formal methods among computer scientists. FM techniques could be used in checking internal consistency of a specification and proving that the system satisfies the desired properties. These characteristics can be automated using computer based tools. FM refers to the use of techniques from formal logic and discrete mathematics in the specification, design, and construction of computer systems and software [23]. The logical properties of a computer systems are described in a form of a mathematical model. This in turn helps the user to rectify requirements of certain properties of system. The foundation of FM are based on reasoning from formal logic. Formal modelling of a system involves transformation of a description of system (non-mathematical model) into a formal specification using formal languages. The mathematical inclusion results in the system description with better logical precision [23]. It also provides better completeness and consistency of system requirements or design.

Definition of Terms:
The following definitions are taken from [23] that depict basic terms required to understand this work.

- A formal specification is a concise description of the behaviour and properties of a system written in a mathematically-based language, specifying what a system is supposed to do as abstractly as possible, thereby eliminating distracting detail and providing a general description resistant to future system modifications. The most formal specifications are written in a language with a well-defined semantics that supports formal deduction and allows the consequences of the specification to be calculated through proof of putative theorems.

- A formal proof is a complete and convincing argument for the validity of a statement about a system description. A proof proceeds in a series of steps, each of which draws conclusions from a set of assumptions. Justification for each step is derived from a small set of rules which state what conclusions can be reasonably drawn from assumptions. Such justification eliminates ambiguity and subjectivity from the argument. Formal proofs may be prepared manually or, preferably, with the assistance of an automated FM tool.

- Abstraction is the process of simplifying and ignoring irrelevant details and focusing, distilling, and generalising what remains. In FM, abstraction is a tool for eliminating distracting detail, avoiding premature commitment to implementation choices, and focusing on the essence of the problem at hand.

2.3.1 Different Formal Verification Techniques

There are numerous formal verification techniques that are based on formal logic and discrete mathematics. Some of them are:

- Theorem Proving
- Model Checking
- Bounded Model Checking (BMC)
- Model-Based Testing (MBT)
- Static analysis with abstract domains
- Equivalence Checking

a. Theorem Proving:
Theorem proving [32] is based on variations of Hoare logic [28] and it can be used for programs involving complex data structures. Theorem proving can also deal with infinite state spaces, as they do not need to exhaustively visit the program state space for verification because it involves constraints on states rather than instances of states. It uses syntactic domain (structural domain), which is much smaller than semantic domain (set of meanings) searched by model checking. It is well suited for data-intensive systems and supports fully automated analysis only in restricted cases i.e. verification of inductive structures can be done through mathematical induction but cannot be automated. Theorem proving has better capabilities than model checkers but the proof of a practical system in theorem proving can be extremely large and require experts with more effort. There has been significant effort to combine the complementary benefits of theorem proving and model checking since past 15 years [32].

b. Model Checking:
It relies on building a finite model [32] of a system and checking that a desired property holds in that model.

$$M \models \Phi$$

An abstract model "M" is constructed in the form of variations on finite state automata and specification formulas Φ is created in the form of variations on temporal logic. A specification or property is a logical formula written either using LTL, CTL or CTL*. To ensure that the formula Φ holds, the verification algorithm involves exploring the set of reachable states of the model. If Φ is an invariant assertion then the model checking approach explores the entire state space to ensure that the formula holds in all states. Hence finite sets of reachable states are required for termination.

Model checking [20] is well suited for "control-intensive" applications (code in form of control structures, "if" statements) on simple data types (e.g. "int" variables). But for complex data types such as trees, lists and recursive definitions it is difficult to verify using model checking. Model checkers are more often applied on C language as it lacks complex data types. Another advantage is that verification can be fully automated and counterexamples are automatically generated if the property doesn't hold true. But this is difficult for the complex structures that use mathematical inductions. One of the drawbacks of this model is that it puts limits on exploring the state space known as the state explosion problem and hence can be considered less robust [32].

c. Bounded Model Checking (BMC):
This technique is a type of Model checking. It performs a depth-bounded exploration of the state space where the BMC explores program behaviour only up to a given depth. Bugs that require longer paths are missed. In BMC [37], the design under verification is unwound "k" times and conjoined with a property to form a propositional formula, which is passed to a SAT (Satisfiability Modulo Theory) solver. This technique is mostly used to find superficial bugs and it provides a full counterexample.

d. Model-based testing (MBT):
MBT is an important technique and approach towards test cases in our work. MBT [36] is software testing in which test cases is generated in whole or in part from a model that describes some (usually functional) aspects of the SUT (System Under Test). It consists of Online MBT and Offline MBT. In offline MBT, models are used for generating conventional test suites that can be later executed on the SUT. In contrast, an online approach constantly executes the tests as they are generated. Increased coverage is seen as especially important in testing concurrent systems that are hard to test using conventional methods. The development time is highly saved because the test model can be created at the same time and we can avoid the rework on wrong implementation. Reusability of the model in long run is the most useful benefit.

One of the advantage of MBT is instead of creating test cases manually, a selected algorithm is generating them automatically from the model [21]. It comprises the automation of black-box test design but white-box testing is also possible. Another advantage of MBT could be that it allows tests to be linked directly to the SUT requirements, which allows readability, mapping and maintainability of tests. And also provides good coverage of all the behaviours of the SUT and to reduce the cost and effort of testing. MBT is a test process that comprises of different test methods that utilise the executable model in MBD as source information. As mentioned in [21] a single testing technique is not sufficient to achieve a desired level of test coverage, different test methods are usually combined to complement each other across all the specified test dimension. If sufficient test coverage has been attained on the model level then designed test cases can be reused for testing the software created based on or generated from the models.

This practice allows the functional equivalence between the executable model, specification and code can be verified and validated. The most generic definition of MBT is testing in which the test specification is derived from both the system requirements and a model that describes selected functional and non-functional aspects of the SUT. The test specification can take the form of a model, executable model, script, or computer program code [21]. The resulting test specifications are executed together with the SUT so as to provide the test results.

e. Static analysis with abstract domains:
Static Analysis techniques [37] are mainly used in compiler optimisation but can also be used for program verification. A sound approximation is necessary in order to inspect all the error present in the program. A subset might miss some of the error while verification and hence we have to consider the superset for guarantees that are not misleading. These techniques can be used to analyse large software systems with minimal user interaction and considered to be extremely robust (can cope with large and varied inputs). But one of the limitations could be that these systems can verify only simple systems.

f. Equivalence Checking:
This approach is mainly applied in hardware design for checking functional equivalence of two similar circuits. It uses Canonical representations, such as Binary Decision Diagrams (BDDs) [1] or Satisfiability Solvers used for comparison. Validate that the implementation of a module is consistent with the specification. It is highly automatic and efficient approach. Most used formal verification technique, primarily in semiconductor industry. Many commercial tools exist such as Design VERIFYer (Chrysalis Inc.) and Formality (Synopsis, million gates in less than an hour).

2.3.2 Benefits and limitations of Formal Methods

For the complexity of software critical softwares it is important that the software should function without any deadlock or errors and this has led to the interest in formal techniques. The following can be effectively addressed by application of Formal Methods [23]:

- Aerospace systems are are mostly software critical, complex in nature and development needs verification techniques.
- Software-Intensive systems are characteristically different from hardware failures.
- Software developed by organisations with higher degree of software quality requires FM techniques to counter remaining defects in the developed product.

Moreover, it is necessary to detect the errors in the software during the development stage to reduce the overall cost of software product and for efficient development of softwares. Some of the benefits of using Formal Methods are listed below [23]:

- Formal specifications holds a high degree of logical precision that removes the ambiguity that is found in informal specifications.
- Formal proofs removes ambiguities in requirement analysis by providing a logical argument about the behaviour of the requirements.
- Formal specification provides a systematic and recursive approach to analysis and thus provides better analysis.
- FM methods can be scaled and tailored to the needs of a project.
- Formal specifications can be used during any life cycle phase. The early use can benefit in detecting defects and efficient software development.
- Formal proofs and specifications can be automated using computer-based tools. This enables the proofs to be re-executed.
- Formal Methods helps to find defects. As shown in [27] the undetected defects were found by using FM that were not evident during extensive testing.
- Formal specifications can also detect errors in requirements and design during the early stages of software development. This helps in reducing mistakes and implementing correct requirements and design.
- Another advantage of using FM techniques could be large coverage of tests cases in finite proof as FM techniques are mainly inclined towards mathematical proofs.

With enormous list of benefits. FM techniques also consists of some limitations. Some of the limitations of FM techniques are listed below:

- Writing formal specifications needs minute attention to details. That is the informal requirements should be transformed into computer understandable language. There could be undetected gaps or deviation of transformation into computer understandable language during the process of transformation due to misinterpretations or erroneous formalization of correctly stated requirements.
- The logical calculations can fail due to a specification or system of equations that does not correctly model the real world. On the other side this could also fail due to misinterpretation of result calculated.
- FM are not well suited for large applications [23]. The size determines the difficulty of using FM, for example the industrial software projects.
- FM are less suited for highly computational applications and numerical algorithms, this makes it not equally similar to all applications. According to [23] the methods can be applied to nearly any applications but it states that higher complexity applications achieve better gains than compared to the lower complexity applications because of the simple fact that less complex problems can be solved using less rigorous methods. Furthermore considering the applications based on mathematical domain including floating point arithmetic, pose some difficulties (working with approximations of real numbers).

2.4 NuSMV Model Checker

NuSMV (http://nusmv.fbk.eu) is designed to allow for the description of FSM. The FSM can range from synchronous to asynchronous and from detailed to abstract models. It provides modular hierarchical descriptions and allows reuse of components. A FSM can be described using boolean, scalar, fixed arrays and static data types. The transition relation of FSM can be described in NuSMV which helps in evolution of the state of the FSM. The description of transitions can be done using propositional expression in propositional calculus [24]. This provides more flexibility in defining the transitions but on the other hand allows inconsistency. The presence of logical contradictions can lead to deadlock. This can make some specifications vacuously true, and makes the description unimplementable [24]. This can be resolved using parallel-assignment syntax.

A simple NuSMV example is as shown in the below example that depicts the status of states between `Ready` and `Busy`. As described above the keyword `MODULE` is used to denote the `main` module. There are three portions of code namely `VAR, ASSIGN` and `SPEC`. `VAR` identifies a portion of code where variables are defined. The `VAR` section defines the set of states. Two variables, each of which has 2 values: in total, 4 possible combinations, i.e., 4 possible states: `s0=(0,ready), s1=(0,busy), s2=(1,ready), s3=(1,busy) ASSIGN` identifies a portion of code where variables are initialised and evolution is described. `ASSIGN` section is used to mainly describe the initial states and its consecutive next states using different `case`. `SPEC` defines properties to be verified

using CTL logic as shown in above section 2.2.2. Similarly we can use LTLSPEC to specify LTL logic as shown in above section 2.2.1. Additionally `FAIRNESS` constraints can be used that restricts the attention only to fair execution paths. During evaluation of specifications, the model checker considers path quantifiers to apply only to fair paths. If the specified `SPEC` is not relevant to the model then a counter example is generated as a finite sequence of transitions through different states and can be represented in a bounded setting as a finite prefix followed by a loop, i.e. a finite sequence of states ending with a loop back to some previous state.

```
VAR
      request : boolean;
      state : {ready,busy};
ASSIGN
       init(state) := ready;
       next(state) := case
                        state = ready & request : busy;
                        1 : {ready, busy};
                        esac;
SPEC
        AG (request -> AF state = busy)
```

NuSMV is used during this work and also during the internship work. The graphical models presented in Chapter 6, 7 and 8 are also formal modelled in NuSMV in the similar methodology.

2.5 Summary

This chapter described the basic concepts of FM techniques, model-checking, requirement elicitation using template based method, formalising requirements using TL and the model-checking tool (NuSMV) used in this work. Firstly, the requirement elicitation process is important to elicitate the requirements in a structured and uniform manner. The drawback of non-uniformity in analytical and natural language requirement method is refined template based requirement method. It explains 6 steps to formulate these requirements i.e. determining the process, activity, legal obligation, refinement, logical and temporal conditions and further expectations to complete these requirements using the SOPHIST rulebook. Furthermore these requirements have to be formulated in a mathematical way i.e. in a computer understandable language. This is done using LTL and CTL temporal logics. These temporal logics becomes the necessity part to develop language that can model the situations and that makes the way to reason about them formally. The propositional logic is based on the propositions that express either being “TRUE” or “FLASE”Ṡome of the examples of LTL and CTL transformations are also provided to explain these temporal logics.

Formal methods provide better completeness and consistency of system require-

ments or design. It can also be used for strict traceability between system descriptions across different life cycle.The benefits achieved from formal techniques are enormous as it involves high degree of logical precision that removes the ambiguity that is found in informal specifications. It can be scaled according to the project needs. Because of its recursive approach it provides better analysis. Another advantage could be that these FM techniques could be used in any life cycle phase in detecting errors and hence makes it flexible to use during any life cycle phase of software development. The automation of formal proofs and specifications using computer based tools enables the proofs to be re-executed. It can also detect errors during early stages of software development and helps in reducing mistakes at early stages and in turn this makes the software development cost effective. The large coverage of test cases helps in finding the absence of errors along with the presence of errors. There are many formal verification techniques some of them mentioned are Theorem Proving, Model Checking, Bounded Model Checking (BMC), Model-Based Testing (MBT), Static analysis with abstract domains and Equivalence Checking. Although Theorem Proving are better capable than model checkers but it requires expert with more effort in proof of practical systems. Although Model checking is well suited for control-intensive applications.

Finally, the model checking tool i.e. NuSMV which is a symbolic model checker is explained with an example. NuSMV allows in describing the transitions using propositional expressions. Another advantage of using NuSMV is it capability to generate counterexamples.

Chapter 3

Related work

The use of formal methods for software verification is described in many articles. Formal methods are well known techniques and in use since 40 years as shown by Pnueli in [33]. Pnueli describes a unified approach to program verification, which is applied to sequential and parallel programs. The paper shows suitable reasoning about concurrent programs.

Recent work by Torens as described in [8] shows the introduction of formal methods and model checking in the test strategy of an automated planning and guidance software module of an Unmanned Aircraft Vehicle. The idea of focus in [8] is to enable the use of formal methods for software testing and validation of requirements, as the certification standard described by aerospace domain such as DO-178C mentions the use of formal methods for software in its supplement DO-333. Under the supervision of Torens, this work takes a similar direction of requirements elicitation and formalization of requirements using temporal logics such as Linear Temporal Logics and Computational Tree Logics, which are used as derivatives of temporal logic. [8] also uses NuSMV as a model-checking tool to analyze the requirements in regards to the model.

In the paper [36] written by Sadhukhan explains the use of Model Based Testing using PIM (Platform Independent Model) in dealing with customer information from the bank data. The system requirements were fed using UML models to Conformiq Qtronic tool and most of the modules were developed in Java.

Another paper [9] by NASA Ames presents model checking and symbolic execution to enable testing of complex softwares by using Java PathFinder model checking tool (JPF) to enable test case generation for Java Programs that have been applied to generate test sequences and test vectors for NASA software.

Mats P. E. Heimdahl and Willem Visser et. al [26] from University of Minnesota and NASA Ames Research Center are working in a similar direction of witnessing counter example capability of model-checkers for constructing test cases for the flight-

guidance system written in the RSML e language. A specification based test generation using NuSMV model-checkers is done along with code based test case generation using Java PathFinder for Java program.

Rayadurgam [4] uses the concept of state-based specifications for generating test sequences. It refers to the model checker to automatically generate complete test sequences that provide structural coverage of specifiec requirements. Simillay, Gargantini [4] describe a method for generating test sequences from requirements specified in the SCR (Software Cost Reduction) notation. To derive a test sequence, a trap property is defined which violates some known property of the specification. In their work, they define trap properties to generate counterexamples.

Beyer [10] uses BLAST as a model checker to automatically generate test suites. In [10] given a C program and a target predicate textitp. The BLAST determines the set of L of program locations which program execution can reach with p true and automatically generates a set of test vectors that exhibit the truth of p at all locations in L

Kendoul in [30] describes the current state of the art in autonomous Rotorcraft UAS (RUAS). It combines the review of last two decades of active research and provides the autonomy level of RUAS using GNC (Guidance, Navigation and Control) aspects. This helps with defining the scope of generality of generic UAV.

Brazier [19] uses a similar approach of generic model and its reuse. It defines a generic agent model (GAM) which abstracts from specific application domains. It works on a similar platform of reusing the generic model as a template or pattern for large variety of agent types and application domain types. The book written by Clements [2] explains the concept of software product lines approach and several benefits in time and cost reduction in development of software.

Chapter 4

Generic and Specific UAV

At this point it is important to understand the concepts of generic and specific UAV as the core functionalities of guidance system is implemented in the FSM. The generic and specific UAV are distinguished based on generic and specific behaviours of the UAV. The high level and low level behaviours of UAV along with the components of GNC and its operations distinguishes the generic and specific UAV.

4.1 Generic UAV

The generic nature of the UAV could be considered at large considering all the UAV present until date but this would again make the idea of generality unclear. Hence the idea of considering the generality is to use ALFURS framework [30] as the ALFURS framework is widely used as a reliable source of information by the RUAS community.

4.1.1 ALFURS Framework

The ALFURS Framework [30] describes the current state of art in autonomous Rotorcraft UAS (RUAS), and provides detailed review of two decades of active research on RUAS. It mainly concentrates on there major parts of UAS i.e. Guidance, Navigation and Control. It provides a standard metrics in characterising and measuring the autonomy levels of a RUAS using GNC as main focus. RUAS has been categorised into five classes based on attributes such as size and payload [30]. A graphical representation of the mentioned categories is depicted in ALFURS framework. The Figure 4.1 [30] represents categories as shown in ALFURS framework.

- Class 1. Full-scale unmanned helicopters or optionally piloted autonomous helicopters. For example Boeing Unmanned Little Bird (ULB) helicopter.
- Class 2. Medium-scale UAS helicopters that are autonomous or semi-autonomous with total weight of more that 30 kilograms. For example Yamaha RMAX.

- Class 3. Small-Scale UAS based on RC-helicopters with optional autopilot integration with total weight of less than 30 kilograms. For example Vario Benzin Trainer.
- Class 4. Mini Rotorcraft UAS that are portable and can fly outdoors as well as indoor environments with total weight ranging from few 100 grams to few kilograms. For example MIT autonomous indoor quad-rotor.
- Class 5. Micro Air Vehicles (MAV's) that are mainly designed for indoor applications. For example Epson micro flying robot.

Figure 4.1: Categories of unmanned rotorcraft system according to ALFURS [30]

4.1.2 Levels of Autonomy

According to ALFURS [30], it defines autonomy as "The condition or quality of being self governing. When applied to RUAS, autonomy can be defined as RUAS's own (own implies independence from human intervention) ability of integrated sensing, perceiving, analysing, communicating, planning, decision-making, and executing, to achieve its goals as assigned by human operator(s) through designed Human-Robot Interface (HRI) or by another system that the RUAS communicates with".The ALFURS levels are categorised based on degree of RUAS engagement with the GNC functions. The autonomy levels is proportional to the level of increase along the GNC functions as shown in the table 4.1 and 4.2.

Level	Level Descriptor	Guidance	Navigation	Control
10	Fully Autonomous	Human-level decision-, making accomplishment of most missions without any intervention from ES (100% (ESI) External System Independent), the cognizant of all within operation range.	Human-like navigation capabilities for most missions, fast SA that outperforms human SA in extremely complex environments and situations.	Same or better control performance as for a piloted aircraft in the same situation and conditions.
9	Swarm Cognizance and Group Decision Making	Distributed strategic group planning, selection of strategic goals, mission execution with no supervisory assistance, negotiating with team members and ES	Long track awareness of very complex environments and situations, interference and anticipation of other agents intents and strategies, high level team SA	Ability to choose the appropriate control architecture based on the understanding of the current situation/context and future consequences.
8	Situational Awareness and Cognizance	Reasoning and higher level strategic decision making, strategic mission planning, most of supervision by RUAS, choose strategic goals, cognizance.	Conscious knowledge of complex environments and situations, interference of self/ others intent, anticipation of near future events and consequences (high fidelity SA)	Ability to change or switch between different control strategies based on the understanding of the current situation/ context
7	RT Collaborative Mission Planing	Collaborative mission planning and execution, evaluation and optimization of multi-vehicle mission performance, allocation of tactical tasks to agent	Combination of capabilities in levels 5 and 6 in highly complex, adversarial and uncertain environment	same as in previous levels (no additional control capabilities are required
6	Dynamic Mission Planning	Reasoning, high level decision making, mission driven decisions high adaption to mission changes, tactical task alloc exec monitoring	Higher-level of perception to recognise and classify detected objects/ events and to infer some of their attributes, mid fidelity SA	same as in previous levels (no-additional control capabilities are required

Table 4.1: ALFURS Autonomy Levels Part 1 [30]

5	RT Cooperative Navigation and Path Planning	Collision avoidance, cooperative path planning and execution to meet common goals, swarm or group optimisation.	Relative navigation between RUAS, cooperative perception, data sharing, collision detection, shared low fidelity SA	Distributed or centralised flight control architectures, coordinated maneuvers.
4	RT Obstacle/ Event Detection and Path Planning	Hazard avoidance, RT path planning and re-planning, event driven decisions, robust response to mission changes,	Perception capabilities for obstacle, risks, target and environment changes detection, RT mapping (optional), low fidelity SA.	Accurate and robust 3d trajectory tracking capability is desired.
3	Fault/ Event Adaptive RUAS	Health diagnosis, limited adaptation, onboard conservative and low-level decisions, execution of pre-programmed tasks.	Most health and status sensing by the RUAS, detection of hardware and software faults.	Robust flight controller, reconfigurable or adaptive control to compensate for most failures, mission
2	ESI Navigation (e.g., Non-GPS)	Same as in Level 1	All sensing and state estimation by the RUAS (no ES such as GPS), all perception and situation awareness by the human operator	Same as in Level 1
1	Automatic Flight Control	Pre-programmed or uploaded flight plans (waypoints, reference trajectories, etc.), all analysing, planning and decision-making by ES	Most sensing and state estimation by the RUAS, all perception and situational awareness by the human operator	Control commands are computed by the flight control system (automatic control of the RUAS 3D pose.)
0	Remote Control	All guidance functions are performed by external systems (mainly human pilot or operator)	Sensing may be performed by the RUAS, all data is processed and analysed by an external system (mainly human)	Control commands are given by a remote ES (mainly human pilot)

Table 4.2: ALFURS Autonomy Levels Part2 [30]

As shown in the table 4.1 and 4.1. It considers mainly 3 systems:

- Flight Control System
- Navigation System
- Guidance System
- Flight Control System: According to [30], RUAS control systems are based on control technologies of manned aerial vehicles. Although RUAS includes additional systems like position/ velocity control, 3D trajectory tracking, heading control etc. As shown in the Figure 4.2, the flight system is classified into 3 main categories. More information is available in [30]. In our case, the flight control system is also considered during modelling of generic information flow model, along with Navigation and Guidance systems.

- Guidance System: The UAS guidance system is a substitute to pilot's deliberative process and decision system. It gives commands to flight controller to attain mission and also importantly safety of mission. Depending on the levels of autonomy the human operator can intervene the ongoing mission. This is clearly mentioned in the ALFUS [29] autonomy levels under "Teleoperation Mode". The ALFUS (Autonomy Levels for Unmanned Systems) defines teleoperation mode as: "A mode of UMS operation wherein the human operator, using sensory feedback, either directly controls the actuators or assigns incremental goals on a continuous basis, from a location off the UMS ". The guidance system is classified into 3 main categories as shown in the Figure 4.3.

- Navigation System: The higher levels of autonomy requires navigation system for sensing, state estimation, environment perception and situational awareness. The ALFURS defines Navigation system as "Navigation is a process of data acquisition, data analysis, and extraction and interference of information about the vehicle's state and its surrounding environment with the objective of accomplishing assigned missions successfully and safely". It is majorly divided into 4 categories as shown in Figure 4.4.

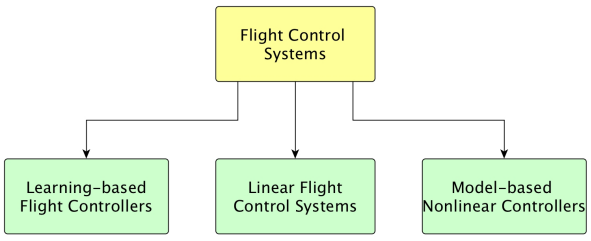

Figure 4.2: Classification of Flight control system [30]

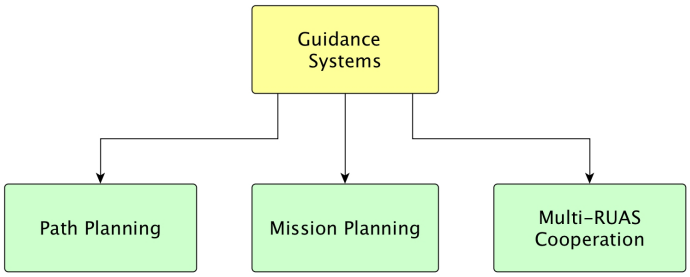

Figure 4.3: Classification of Guidance system [30]

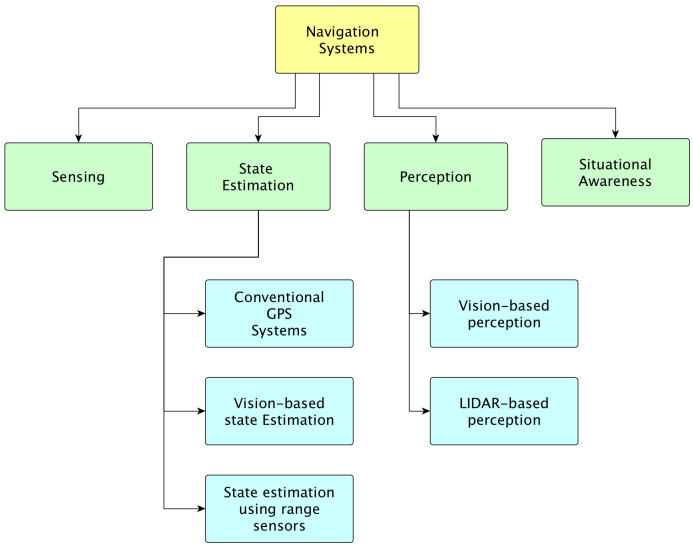

Figure 4.4: Classification of Navigation system [30]

4.2 Specific UAV

The specific UAV considered in our case is the ARTIS UAV. The DLR Institute of Flight systems in Braunschweig develops the ARTIS platform for the research in autonomous behaviour of UAV's. The detailed description of the ARTIS UAV is explained in the next section.

4.2.1 The ARTIS Platform

The ARTIS platform consists of a fleet of UAV's for research on autonomous flight. It ranges from a fixed wing UAV called "Prometheus" (as shown in Figure 4.5) to several rotorcraft UAV's. The latest addition to the ARTIS fleet is capable of maximum take-off weight of 150 kgs named "SuperARTIS" with 3 meter rotor diameter [6] (as shown in Figure 4.6).

The ARTIS framework can operate within ALFURS level 4 to 6. It has the capability to be operated under[14]:

- Mission Mode: In this mode the predefined mission is executed to complete a given task. It can operate in online or offline modes subsidiarily. The UAV is assigned a mission with all the relevant information needed to achieve this task. In case of the offline mode a 3D World information has to be loaded along with the task and other parameters like vehicle dynamics. But while in the online mode the 3D World model is generated by itself by using the vision services.
- Command Mode: In this mode the MissionController is receiving direct commands from the operator at ground station. The direct commands can signify the change in the mission. The command mode could intervene the mission mode through operator intervention.
- Remote Control Mode: In this mode the remote pilot gives direct flight controls to the UAV. In this mode the "MissionControllerOff" state is active. It signifies that the MissionController is not operating and is turned off. Here a ground based operator can control the helicopter using a remote control.

Figure 4.5: Prometheus Fixed Wing UAV [www.dlr.de]

Figure 4.6: SuperARTIS Rotorcraft UAV [www.dlr.de]

The research idea towards ARTIS platform is the implementation of decision making components for onboard mission management. The main goal is to enable onboard system which allows different levels of UAV autonomy for the urban areas [14].

4.2.2 MiPlEx System

The MiPlEx function is used onboard in the unmanned helicopter ARTIS. It performs online and offline path planning and execution and mainly concentrates on the urban terrain where the buildings, street poles, trees, birds are closely packed and this poses the major challenges to avoid obstacles and achieve the target mission. It is implemented onboard of unmanned aircraft vehicle ARTIS designed by DLR institute of Flight Systems. Unlike ALFU(R)S generic unmanned model the Guidance function is implemented within the MiPlEx and is capable of receiving the inputs and commands from the *Operator/ Human Element* on ground. In offline planning, the *operator* can send necessary preloaded information's such as the task sets, 3D World Model and other parameters (dynamic limits of the specific aircraft) to the aircraft. It has a fault tolerant capability by having self-sufficient decision taking potential during low fuel level and monitors communication links between the *Mission Manager* and *Operator*. The Figure 4.7 shows the abstract model of the MiPlEx states specifically in the offline mode. It is based on the deterministic roadmap path planner [9] that is combined with the real-time obstacle mapping algorithms.

The *Mission Planner* generates the *Mission Plan* or the *Behaviour Sequence* and forwards it to the *Mission Manager* for generation of flight controls. The system is capable of working in offline as well as online mode. In case of the offline mode the *user sequences* are uploaded into the system such as the *3D World Model* as shown in the Figure 4.7 and in case of online planning the system has to project its own optimum route using real-time functionalities. The mission manager involves two subdivisions mainly "Supervisor" and "Sequence Controller".

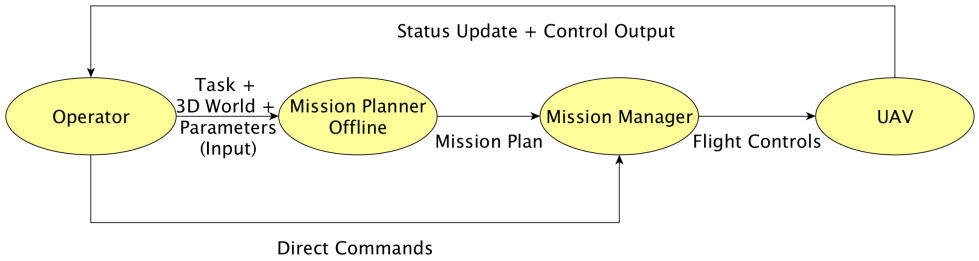

Figure 4.7: The Abstract MiPlEx Offline Model

Supervisor and Sequence Controller

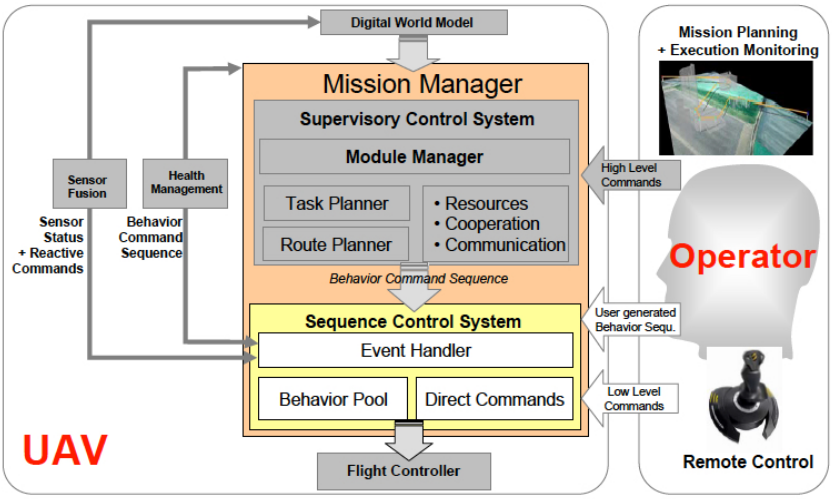

Figure 4.8: The ARTIS Sequence Control System [14]

As shown in the Figure 4.8 that describes the component organisation of the Mission Manager. As shown in the Figure 4.8 the Supervisor operates on a higher level and monitors or influences the control on the Sequence Controller. The Supervisor is mainly responsible to handles/ concise of higher level commands like *"Gate Mission"* or *"Search and Rescue"* whereas the Sequence Controller handles/ concise of low level behaviours such as *"HoverTurn"* or *"TakeOff"* as shown in Figure 4.9 [14]. This is also shown in Figure 4.9 in terms of a 3T architecture [14] that mainly consists of 3 layers:

- Reactive Layer
- Sequencing Layer

- Deliberate layer

The low level behaviours are located at the reactive layer (also called as skill layer) while the higher level behaviours are located at the deliberate layer. The reactive or skill layer consists of 2 basic behaviour groups. The former group consists of behaviours having direct position and velocity commands to the flight controller, for example the take-off and landing behaviours uses a fixed position for horizontal layer on X and Y axis and a velocity command on the Z axis. These type of behaviours are called reflexive behaviours. The second group has the largest set of behaviours that consists of trajectory-based control commands. These behaviours contains behaviours to wait at a position (WaitFor), turn on the spot (HoverTurn), fly along a linear trajectory towards a location (HoverTo), to fly around a point along a horizontal circular trajectory (Pirouette), and to do fast forward flight along an arbitrary trajectory (FlyTo).

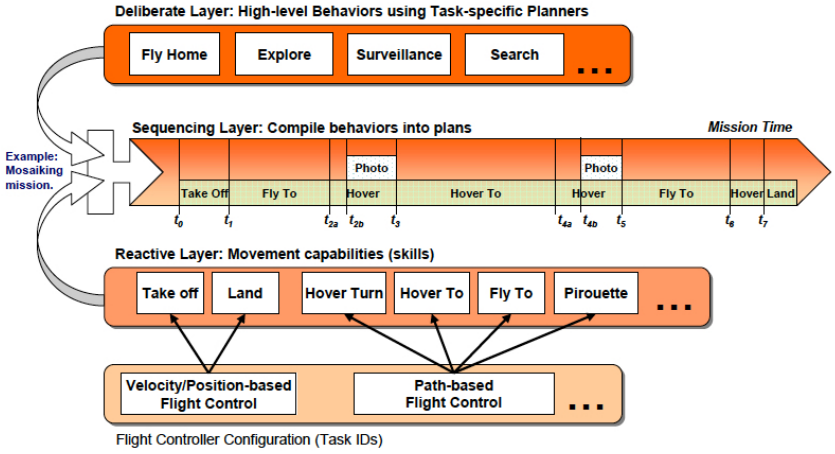

Figure 4.9: 3T sequencing layer of ARTIS Sequence Control System [14]

The deliberate layer consists of complex behaviours. These behaviours are capable to change the current ongoing mission. However, the compilation can take place onboard as well as by the ground control station and sent directly to Sequence Control System.

According to Figure 4.8, the Sequence Control System can interface to the flight control system. The Sequence Controller can inputs from many sources like the ground control station, vision computer this enables the Sequence Controller to provide safe handling and robust coordination [14].

4.3 Summary

The generic and specific UAV are distinguished based on specific behaviours of the UAV. The generality of the generic UAV is within the scope of ALFURS framework, which is one of the reliable sources in the RUAS community. The ALFURS framework describes the current state of art in the autonomous Rotorcraft UAS and provides detailed review of active research on RUAS. It concentrates on three major parts of UAS i.e. Guidance, Navigation and Control. The ALFURS framework defines the autonomy levels of UAS depending on its autonomous capabilities. It categorises UAS into five categories depending on size and payload capabilities from Class 1 to Class 5. The autonomy levels in ALFURS framework is categorised from "Level 0" to "Level 10". Level 0 being Remote Control that is operated by remote pilot whereas Level 10 being Fully Autonomous that is capable of human level decision making capabilities.

The specific UAV on the other hand is the ARTIS UAV which is developed by the DLR Institute of Flight systems in Braunschweig. The main purpose of the ARTIS platform is for the research in the autonomous behaviour of UAV's. It consists of several fleets of UAV's ranging from fixed wing to rotorcraft UAV's. The ARTIS UAV can operate in 3 modes namely Mission Mode, Command Mode and Remote Control Mode. The "Mission Mode" executes the predefined mission to complete a given task whereas the "Command Mode" receives direct commands from the operator at ground station. These direct commands can change the current ongoing mission or create a new mission altogether. In the "Remote Control Mode" the UAV is under the influence of the remote pilot. In this mode the guidance system of UAV is switched off as all the guidance functions of the UAV is carried out by the remote pilot. The MiPlEx (Mission Planning and Execution) defines the guidance system of the ARTIS UAV. It can operate under online and offline planning modes. In the offline mode, the operator can send necessary preloaded information's such as the task sets, 3D World Model and other parameters (dynamic limits of the specific aircraft) to the aircraft. In this mode the user sequence are uploaded into the system such as the 3D World Model and in case of the online planning the system has to project its own optimum route using real time functionalities.

The MiPlEx consists of "Mission Planner" and "Mission Manager". The Mission Manager is responsible for execution of the generated mission plans by the Mission Planner. The Mission Manager consists of the "Supervisor" and "Sequence Controller System". The Sequence Controller consists of the "3T architecture" consisting of 3 layers namely Reactive layer, Sequence layer and Deliberate layer. The deliberative layer consists of the high level behaviours whereas the reactive layer consists of the low level behaviours. These low level behaviours are categorised into two parts namely Velocity/ Position-based Flight Control and Path-based Flight Control.

The knowledge about the construction and components of the MiPlEx is very significant in understanding the finite state model of specific UAV whereas the ALFURS framework is necessary for the understanding of the generic UAV.

Chapter 5

Contribution to this Master Thesis

This Chapter mentions the contributions towards the Master thesis in brief. Although detailed aspects of contributions and technical aspects of work are discussed in the later Chapters 6, 7 and 8.

5.1 Requirement Extraction and Formulation

Requirements extraction is necessary process and artefact in the process of validation of software. The sources considered for gathering requirements could be from customers for example. In this work the requirements gathered are mainly divided into 2 sets:

- Generic Requirements
- Demonstrator/ ARTIS Specific Requirements

The Generic requirements caters to requirements that are common amongst set of UAV's categorised under the ALFURS by Kendoul [30]. It describes the art in RUAS, and comprises of detailed literature review of last two decades of active research on RUAS. Some of the requirements for a fixed wing UAS are also considered from ALFUS (Autonomy Levels for Unmanned Systems) framework edited by Huang [29] with contributors ranging from U.S Air Force Research Laboratory et al. ALFUS gives a more generic approach towards unmanned systems in general. It also provides terminology and definitions that serve as common reference for aerospace community.

Some of the requirements were also considered from STANAG (Standardization Agreement) [31], as it gives better information about UAV control systems. STANAG was developed to have close co-ordination and the ability to quickly task available UAS (Unmanned Aerial Vehicle Systems) assets.

These documents provide enough knowledge to understand the generic behaviour of the UAS systems.

Furthermore the ARTIS specific requirements are mainly extracted from the source code, core software developers and its documentations. The source code is implemented in C++ and it is possible to extract requirements from the source code by understanding the code itself and also some of the comments marked within the code.

Several interactions and brainstorming sessions with Christoph Torens (Thesis Supervisor, Research Scientist at DLR and AIAA Member) and Florian-Michael Adolf (Research Scientist at DLR and AIAA Senior Member) helped this work to extract and understand the specific nature of ARTIS UAV's. Also suggestion and query solving by Lukas Goormann (DLR) has evaded ambiguities in understanding the ARTIS platform better.

The documentations and technical papers written by Adolf in [14], [13], [16], [15], [17] and [18] along with technical papers by Torens in [8], [6] and [7] also caters to extraction of ARTIS specific requirements. Along with the above mentioned sources, some of the requirements are taken from the previous internship work collaboratively extracted by Girish Patil and the author of this work. Additionally, Christoph Torens has provided with some of the requirements.

All the requirements are formulated using Template Based requirements [34] because of the ambiguities present in understanding requirements from Natural Language and Analytical approach. The analytical approach and template based requirements elicitation are mentioned in Chapter 2 in detail.

5.2 Requirement Formalisation

The requirements formalisation is a process of converting the human understandable requirements i.e. written in simple english language to mathematical form using temporal logics. The following two temporal logics were considered in requirements specification as explained in Chapter 2.

- Linear Temporal Logics (LTL)
- Computational Tree Logic (CTL)

Temporal Logic extends propositional or predicate logic by modalities that permit to referral to the infinite behaviour of a reactive system [5]. They provide mathematical precise notation for expressing properties of the system. The transformation of requirements is necessary for machine understandable language. While LTL considers only linear time in sequence, CTL logic consists of tree and branching of time.

5.3 Extraction of Transitions from Source Code

The MiPlEx consists of several C++ classes within the code. It consists of two classes namely "Supervisor.cpp" and "SequenceController.cpp" which are the major part of focus of this work. The MiPlEx is explained in detail along with Supervisor and Sequence Controller in Chapter 4.

The main idea behind extraction of transitions from source code is because it caters to ease of designing finite state machine and also gives a organised list of transitions between different states. A tabular description is given in the Chapter 7 of this work. This also helps as a bridge between FSM and the source code, providing a direct mapping between the two of them.

5.4 Generic FSM of UAV Guidance System

The generic FSM of UAV guidance system is realised using the generic requirements and complying to ALFURS autonomy levels. The ALFURS autonomy levels ranges from 0 to 10. For example 0 being the UAV under remote control operation to 10 being fully autonomous. The designed system in this work complies within ALFURS levels 0 to 6 i.e. it is capable of handling functionalities from being operated by a remote pilot to making some decisions on its own like "Dynamic Mission Planning" under which the UAV is capable of reasoning, high-level decision making, mission driven decisions, high adaption to mission changes, tactical task allocation and execution monitoring.

Firstly, the information flow model consists of the entire GNC components of UAV and describes the information flow along all the GNC components. It can be mapped to ALFURS levels 0 to 3 and ALFUS teleoperation mode. The generic information flow model was created during the internship work. It has be optimised and mapped to the GAM to better reflect the system during this thesis work.

Secondly, a behaviour model of generic model is created which is focused to guidance system that reflects mission manager operations. It can be mapped to ALFURS levels 0 to 6 and ALFUS teleoperation mode to comply to generic behaviour of the system.

5.5 Specific FSM of UAV Guidance System (MiPlEx)

The specific finite state model of UAV guidance system consists of two finite state models functioning independently i.e. Supervisor FSM and Sequence Controller FSM. The two components together constitutes mission manager in total as shown in Chapter 4. The specific FSM is mapped up to ALFURS level 6 autonomy along with ALFUS teleoperation mode. This is one approach of using the generic approach in designing

the FSM as the specific FSM can be mapped to a generic framework.

In this work the difference between the specific and generic model is the implemented low level and high level behaviours and the absence of slowdown state. The low and high level behaviours are specific to UAV's operations and capabilities.

5.6 Model Checking using NuSMV

NuSMV (http://nusmv.fbk.eu) is a symbolic model checker, As mentioned in the given link it uses aiger 1.94 (http://fmv.jku.at/aiger/) for verifying properties. The description of NuSMV coding is explained in Chapter 2 sections. The generic and specific FSMs of UAV guidance systems are coded into NuSMV and run using textitEclipse IDE.

Firstly the states used in the model have to be defined that involves all the states. Secondly, the initial states has to be specified so that the system understands the initial condition of the entire states from where the transition has to begin. The state transitions have to be specified according to the model using the *next* functionality. The *FAIRNESS* constraints has to be specified to each states such that every state is visited at least once with this functionality. Finally, requirements are formalised using CTL and LTL temporal logics and specified within the NuSMV code as shown in Figure 5.1. The model checker then checks the specified property against the system.

After compiling the described model and running it using *Eclipse IDE* we get a list of temporal logic that satisfies and some that doesn't satisfy the model. A counterexample is generated if the specification is FALSE, this could be the reason that the model fails to satisfy a property serving a indispensable debugging information. The counterexample indicates how the model could reach the undesired state. It describes an execution path that leads from the initial system state to a state that violates the property being verified. With the help of counterexample, the user can trace the violating transition or state, in this way obtaining useful debugging information, and adapt the model (or the property) accordingly.

For the one that doesn't satisfy the model there can be several reasons. Firstly the TL has to be specified more correctly or there has to be certain changes made to the model in order to make the model satisfy all the requirements or in some cases the requirement itself has to be changed as shown in the Figure 5.2.

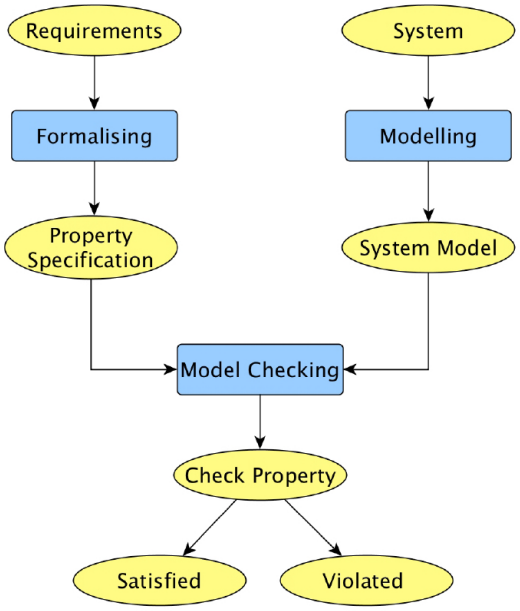

Figure 5.1: Model Checking approach

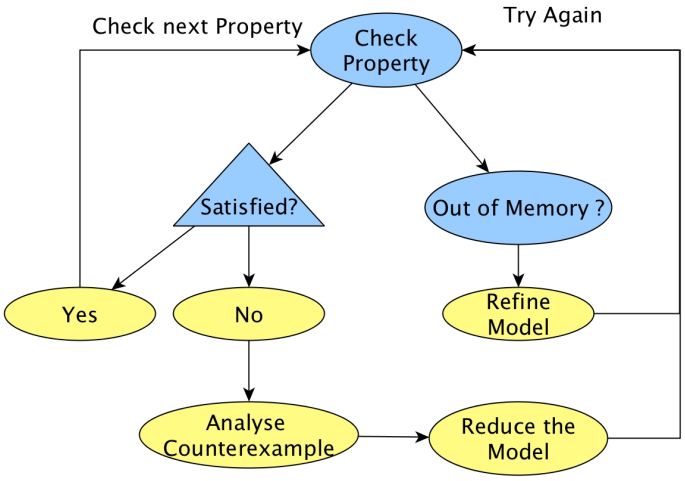

Figure 5.2: Analysis Phase

5.7 Generating Counterexamples for Test Generation

As discussed above a model checker is used to analyse a FSM for property compliance and violations. If the model checker detects no violations, then the property holds TRUE. On the other side a counterexample is generated incase there is a violation of the property against the system behaviour. The use of trap properties according to [4] forces the model checker to generate counterexamples. These counterexamples are generated intentionally to construct test sequences. A trap property is a negation of a property that is TRUE otherwise.

5.8 Approach towards Test Cases

The generated counterexamples itself serves as the bases for test generation. The test cases are considered by using trap properties to all the requirement specification that are TRUE otherwise. A mission scenario is considered from the log files. The mission description is described in Chapter 8. This enables or gives a scenario under which the system is tested. Similarly, a FSM of mission scenario is designed and followed by model checking using NuSMV to describe the test scenario.

The detailed work and contribution to the thesis is shown in the following Figure 5.3.

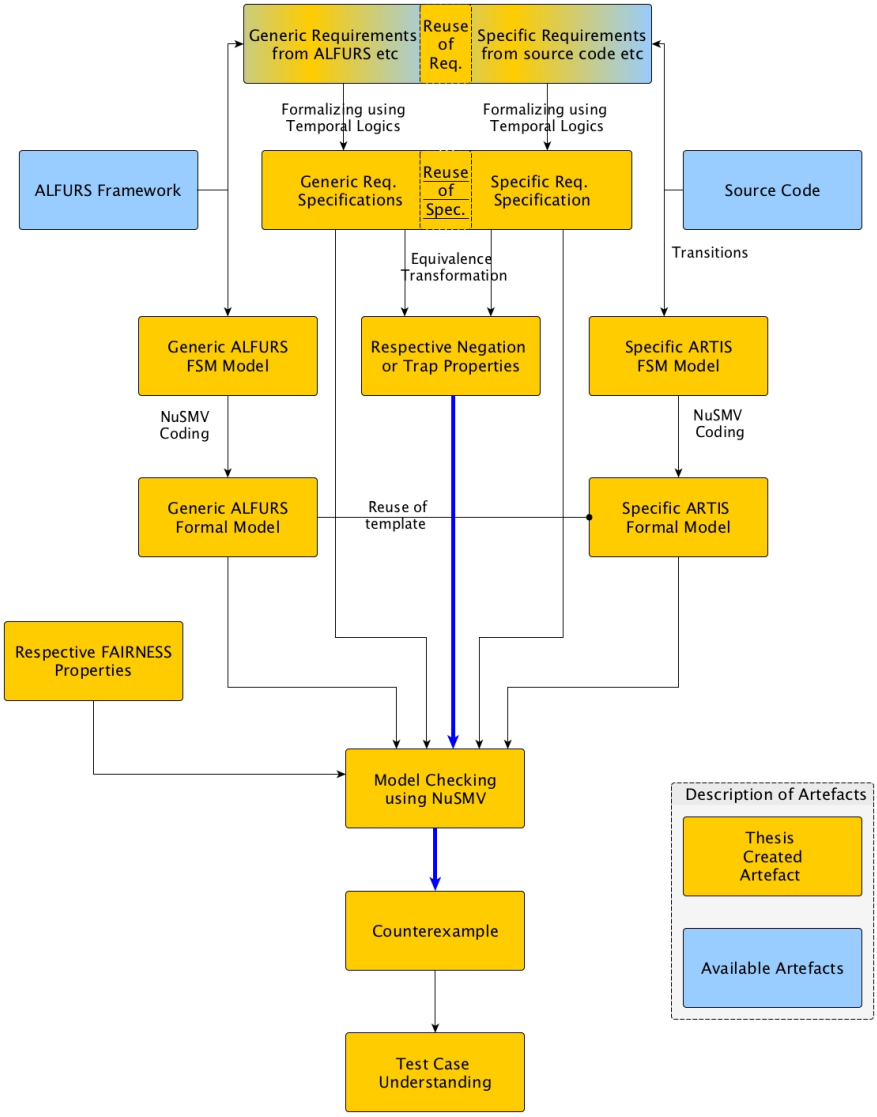

Figure 5.3: Thesis Contribution Summary

Chapter 6

Generic UAV Finite State Machine

This Chapter describes the process of model-checking of generic UAV guidance system. The generic FSM is created using the generic requirements and complying to ALFURS autonomy levels. The ALFURS autonomy levels ranges from level 0 to level 10. The major component in guidance system is Mission Manager. The mission manager is a substitute to pilot's deliberate process and decision system. It includes several high level and low level behaviours of the system.

The following set of steps need to be followed in the process of generic model checking. Firstly, it is important to extract requirements that comply with the generic UAV system because this is also useful in designing the FSM. These requirements need to be formalised in a mathematical way to make it computer understandable. Finally, the system need to be described in NuSMV model checker along with the formalised requirements to know if the requirements matches the system behaviour. In the last step it is necessary to generate the counterexamples from the satisfied specifications using the trap properties.

The process steps is as given below.

- Requirement Analysis
- Formalising Requirements
- Designing of FSM for generic guidance system
- Formal Modelling using NuSMV model checker
- Formal Verification
- Counterexample generation using trap properties

In the next section, the generic modelling approach is explained using information flow diagram of generic GNC component created during the internship work. During the thesis work the information flow diagram is refined and mapped to the GAM [19]. Instead of designing a new model from scratch every time we can make use of the generic model. A specific model is a model with more specific processes, at low level of process abstraction. As explained by Brazier in [19] the process of instantiation could be used for refinement at lower level of knowledge abstraction.

6.1 Generic Modelling Approach using Generic Information Flow of GNC

A model can be considered as generic in two cases [19]. Firstly, it could be "generic with respect to the processes or tasks"
or it could be "generic with resect to the knowledge structures". The former refers to the level of process abstraction i.e. a generic model abstracts from processes at lower levels. In specific sense this means that specific model represents processes in a model with more specific processes, at a lower level process abstraction. Considering the later i.e.. genericity with respect to knowledge refers to levels of knowledge abstraction. A generic model abstracts from more specific knowledge structures. Although the former technique of using genericity with respect to "processes or tasks"
is mainly used in the work.

As shown in the Figure 6.1 [19]. It depicts the process modelled within the generic agent model. It categorises process into "Own Process Control", "World Interaction Management" "Agent Interaction Management" "Maintenance of World Information" "Maintenance of Agent Information" "Co-operation Management" and "Agent Specific Task".

The process involved in controlling an agent for example monitoring its own goals along with processes of maintaining a self-model are task of own process control. The processes involved in managing communication with other agents are the task of agent interaction management. The processes involved in managing interaction with the external World are the tasks of World interaction management. The processes involved in maintaining knowledge of other agents knowledge is the task of maintenance of agent information. The process involved in maintaining knowledge of external World is the task of maintenance of World information. The processes involved in co-operation management involves all the tasks related to social processes, that is co-operation in a project. The processes involved in specific tasks for which a agent is designed is considered in agent specific tasks.

As shown in Figure 6.2 taken from Brazier [19], it depicts the interface information types of components within the agent. These are based on the internal primitive agent concepts[19]. The component own process control uses belief information on other

agent and external world, as input. The output of the component own process control is the agent's characteristic used by other processes. Similarly other process interfaces are explained in more detailed in [19].

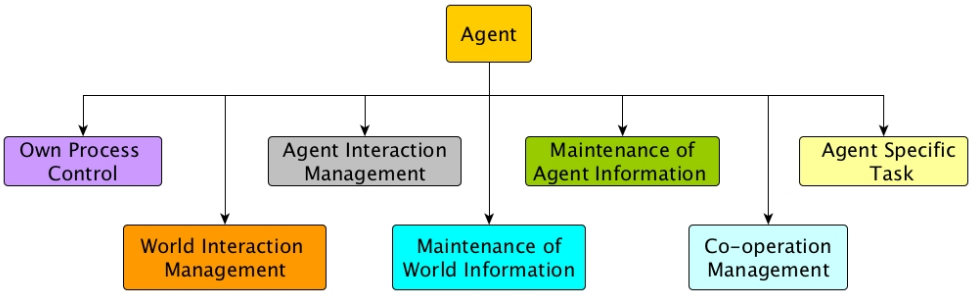

Figure 6.1: Processes within the Agent [19]

Our "Generic Information Flow Model" can be mapped to the Generic Agent Model concept. This brings a sense of generality to our model and the concept of reusing a generic model. As shown in Figure 6.3 the information flow model includes the "Operator Control Unit", "Guidance System", "Navigation System", Remote Control System", "Control System", "UAV State" along with the "Health Monitoring" and "Data Link Monitoring System". The detailed description of the states and edges within these systems is explained in Tables 6.1 and 6.2 . The information flow

model is also mapped to the ALFURS framework until level 3 along with the ALFUS Teleoperation Mode. Different colour codes are used to represent these mappings to the agent processes. For example, the "outCmds" state within "Operator Control Unit" maps to the "Agent Interaction Management" because this state manages its communication with other agents such as "inCmds". Although the "outActCmds state" within "Control System" involves the component of two processes i.e. "Agent Interaction Management" and "Co-operation Management".

The book "Software Product Lines" written by Clements [2] describes the benefits of reusing the software product in terms of organisational and timely benefits. Suppose if we consider a scenario of building a control system for different UAV types ranging from fixed wing, rotorcraft and quadracoptors. Building each of the system from the scratch would be very difficult for each of these UAV types and it would involve more resources and time. Therefore Clements explains the concept of software product lines through which the commonalities between different system could be recognised and only the implementation differences can be realised.
Similarly we can make use of the same concept in our case in generic modelling. This would mean that designing further specific cases would be realising only the differences in modelling the generic and specific models. This enables us to save time and also resources.

CelciusTech (complete case study is available in [22] by Brownsword) launched a product line effort to build their software. A product line is a set of products that together address a particular market segment or fulfil a particular mission. A product line succeeds because the commonalities shared by the software products can be exploited to achieve economies of production. A software product line is a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. The product lines are economical because each product is formed by taking applicable components from the base of common assets, tailoring them as necessary through preplanned variation mechanisms such as parameterisation or inheritance, adding any new components that may be necessary, and assembling the collection according to the rules of a common, product-line-wide architecture. Building a new product (system) becomes more a matter of assembling or generalisation than one of creation; the predominant activity is integration rather than programming. For each software product line there is a predefined guide or plan that specifies the exact product-building approach. Instead of designing each and every new application from the scratch.

Benefits of Generic Modelling:
The generic modelling approach avoids designing of a new specific model from scratch and allows significant reduction in time, expertise and effort. Some of the benefits of using generic modelling approach are listed below:

- Extensive requirements analysis is saved. Some of the requirements can be reused.
- With respect to the modelling and analysis with each new product, time required in designing is reduced.
- In case of testing, new product the 'time to market' problems have reduced and synchronisation, and absence of deadlock have been eliminated. This reduces the effort in testing significantly.
- With the reuse of requirements the effort of converting them into formal specifications is also reduced as the reuse of requirements directly reduces the time in converting of template based requirement into formal language.
- A similar structure of finite state model could be used to resemble the generic case. This gives better understanding of the system and its operation.
- On the organisational front it gives better time-to-market and product quality.

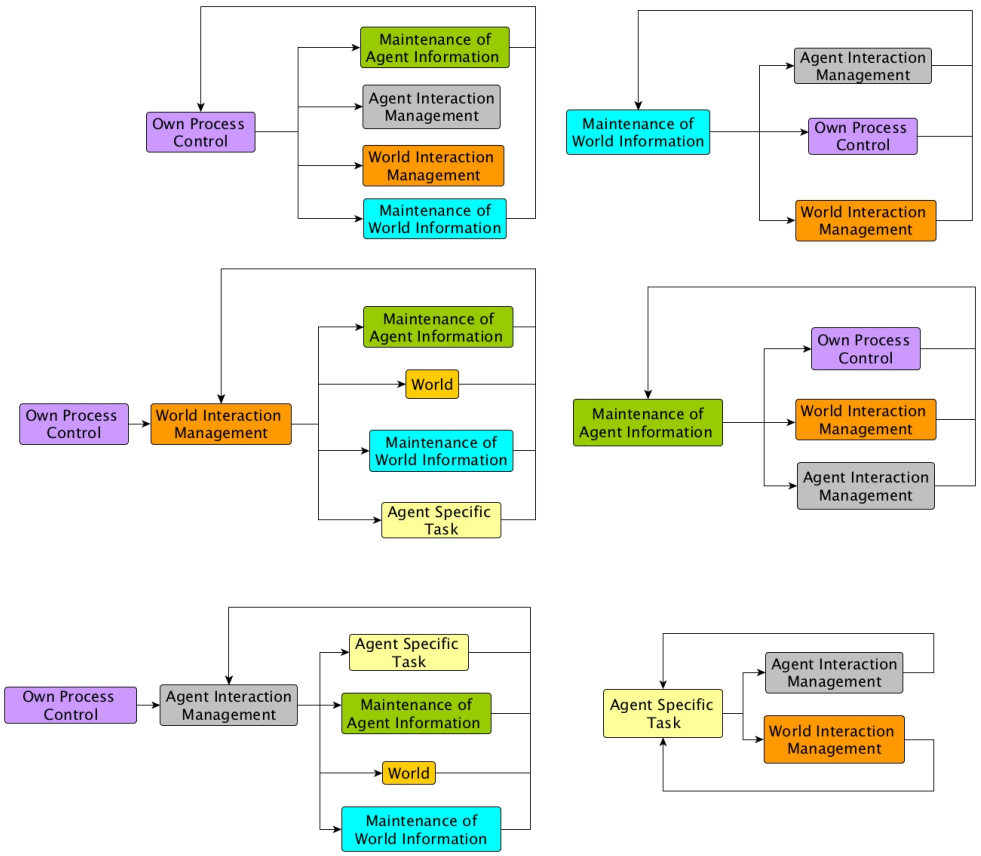

Figure 6.2: Agent Model Local Neighbourhood [19]

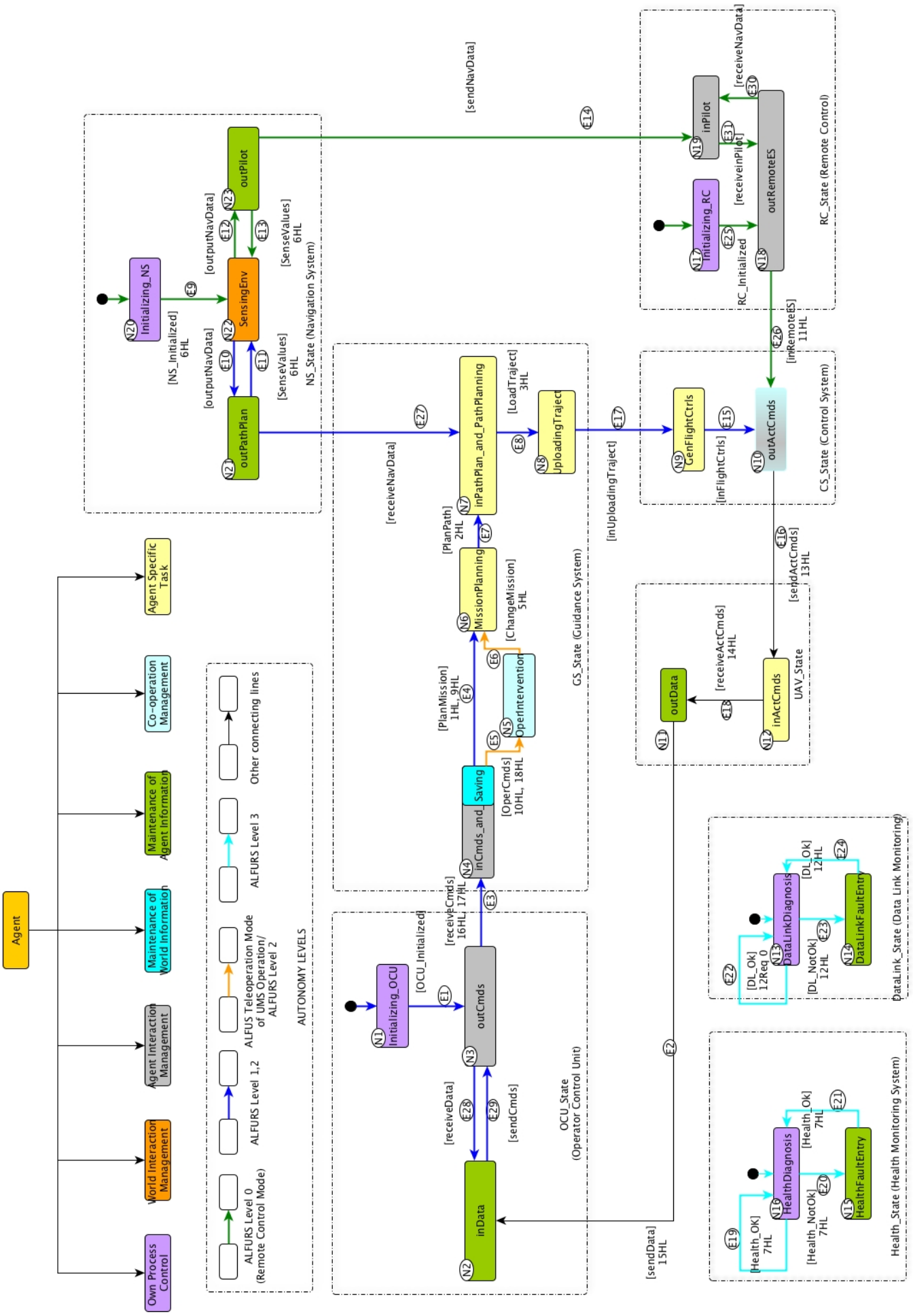

Figure 6.3: Information Flow Mapped to Agent Model

6.2 Generic Requirements Elicitation and Formalization of UAV Guidance System

The generic requirements are realised keeping in mind the commonalities of specific UAV's. The scope of the generality lies within the ALFURS framework So, to cater to this difficulty it is necessary to follow a framework which considers and describes the current state of art in autonomous UAS. The ALFURS framework [30] provides a detailed review of two decades of active research on RUAS. As mentioned in Chapter 4, it categorises the UAS into 5 classes depending on the size and autonomous behaviour and plots the autonomy levels depending on the Flight Control System, Navigation System and Guidance System. Our point of interest is to extract the requirements for the guidance system mentioned in the ALFURS framework. Some of the missing gaps of requirements extraction are filled using the STANAG (Standardization Agreement) [31] and ALFUS [29]. The STANAG gives better information about UAV control system. STANAG was developed to have close co-ordination and the ability to quickly task available UAS. The ALFUS framework provides better terminology and autonomous behaviour about generic unmanned systems in general irrespective of only UAS. These 3 documents provide good understanding about the generality about the generic UAV.

The following table 6.3 provides some of the requirements gathered during our work. The requirements are formulated using template based method which is also called as semi-formalised specification method.The "Logical/ Temporal Conditions" define of the requirement should be valid during all the time or followed by another activity. The "System under focus" defines the validity of the requirement to particular system or in general. The requirements are categorised into high level (HL) and low level (LL) requirements based on "WHAT" and "HOW". The high level requirements describes the intended software functionality using the "WHAT" criteria to describe the requirements. The low level requirements describes the "HOW" the software shall execute the designated functionality.

The formalization of requirement done using LTL specifications. An example of Requirement 1HL (from Table 6.3) is as formulated below:

LTLSPEC G ((MM_State = HumanOperated) -> F(MM_State = MissionExecution));

In the above LTL specification the Mission Manager is code named as "MM_State" for the ease of coding the system behaviour in NuSMV.

Abbreviations	Expansions	Meaning
Operator Control Unti States (OCU_State)		
Initializing_OCU	Initializing Operator Control Unit	Initializes the Operator Control Unit
outCmds	Output Commands	OCU sends the commands to Guidance system from this node
inData	Input Data	The control outputs from the UAV is received by this node
Operator Control Unit Commands		
OCU_Initialized	Operator Control Unit Initialized	This Command ensures the proper initialization of OCU and thereby transition to operation
receiveData	Receive Data	Directs the OCU to receive the Control outputs from UAV
sendCmds	Send Commands	Directs the OCU to send the commands to the Guidance System
receiveCmds	Receive Commands	Directs the "outCmds" node to send the commands to Guidance System
Guidance System States (GS_States)		
inCmds_and_Saving	Input Command and Saving	Receive the Commands sent by OCU and save the received commands
OperIntervention	Operator Intervention	This node intervenes the ongoing mission plan or can also specify new mission to the Guidance System
MissionPlanning	Mission Planning	The process of generation of a Mission Plan takes place at this node
inPathPlan_and_PathPlanning	Input to Path Plan and Path Planning	This node receives the Navigation data and does Path Planning accordingly
UploadingTraject	Uploading Trajectories	In the Offline Mission Planning the trajectories are loaded whereas in online mission planning the Trajectories are generated
Guidance System Commands		
PlanMission	Plan Mission	After receiving the required data the "PlanMission" directs to the actual process of Mission Planning
OperCmds	Operator Commands	Denotes the Commands sent by the Operator
ChangeMission	Change Mission Plan	Denotes the interruption caused to the current mission plan or the process of uploading a new mission plan directly by the operator
PlanPath	Plan Path	Triggers the Path Planning process
LoadTraject	Load Trajectories	Triggers the uploading of trajectories
inUploadingTraject	Input Uploaded Trajectories	Sends the trajectories to Flight Controls
Control System States (CS_State)		
GenFlightCtrls	Generating Flight Controls	Receives the trajectories from "UploadingTraject" and generates the flight controls
outActCmds	Output Actuator Commands	Sends the Actuator Commands to UAV
Control System Commands		
inFlightCtrls	Input Flight Controls	Makes the "outActCmds" to receive the generated flight controls
sendActCmds	Send Actuator Commands	Sends the Actuator Commands to the UAV
UAV State (UAV_State)		
outData	Flight and Output Data	Sending the data to the OCU
inActCmds	Input Actuator Commands	Receives the actuator commands sent by the Control system and directs it to the actuators
UAV Commands		
receiveActCmds	Receive Actuator Commands	Triggers the "Flight_and_outData" to receive the actuator commands
sendData	Send Data	Sends the control outputs to the OCU

Table 6.1: Description of states Part 1

Abbreviations	Expansions	Meaning
Navigation System State (NS_State)		
Initializing_NS	Initializing Navigation System	Initializes the navigation system
SenseEnv	Sense Environment	Included the sensing and state estimation of altitude, position and velocity of the environment
outPathPlan	Output to Path Plan	Sends the navigation data to the path planning process during the Semi Autonomous and Teleoperation Mode
outPilot	Output to Pilot	Sends the navigation data to the pilot during the Remote Control Mode
Navigation System Commands		
NS_Initialized	Navigation System is Initialized	Triggered when the Navigation system is initialized correctly
outputNavData	Output the Navigation Data	Send the Navigational data to the respective recipients
SenseValues	Sense Values	Redirects the Navigation System to Sense the Environment
receiveNavData	Receive Navigational Data	Triggers the Navigational data sent to the Path Planning process
endNavData	Send Navigational Data	Triggers the Navigational data sent to the Pilot
Remote Control System States RC_State		
Initializing_RC	Initializing Remote Control System	"Turn on" of the Remote Control switch
outRemoteES	Output Remote External System Commands	Commands from the Remote Controller
inPilot	Input to Pilot	This node receives the Navigational Data
Remote Control System Commands heightRC_Initialized	Remote Control System Initialized	Triggered when the Remote Controller is turned on correctly and under full operation
receiveNavData	Receive Navigation Data	Triggered to receive Navigational Data
receiveinPilot	Receive from "inPilot" Node	receives the values from "inPilot"
inRemoteES	Input from Remote External System	Send the "RemoteES" Commands to the Control System
Health Monitoring System States (Health_State)		
Health Diagnosis	Health Diagnosis	Monitors the health status
HealthFaultyEntry	Health Fault Entry	Records and generates the error report of Health faults
Health Monitoring System Commands		
Health_Ok	Health Status Ok	Monitored Health status is good
Health_NotOk	Health Not Ok	Monitored Health Status is not under safe tolerance
Data Link Monitoring System States (DataLink_State)		
DataLinkDiagnosis	Data Link Diagnosis	Monitors the Data Link Communication Status
DataLinkFaultEntry	Data Link Fault Entry	Records and generates the error report of Data Link Communication faults
Data Link Monitoring Commands		
DL_Ok	Data Link Status Ok	Monitored Data Link status is good
DL_NotOk	Data Link Status Not Ok	Data Link is interrupted or no clear communication between OCU and UAV

Table 6.2: Description of states Part 2

Req. ID	Logical/ Temporal Conditions	Precondition	System under focus	Legal Relevance	Activity	Whom?	What?	Process Word	Additional Details
1HL	Always		THE GUIDANCE SYSTEM	SHALL	provide the ability	to the Human Pilot	to switch	between Human Pilot Mode and Mission Mode	
2HL	Always		THE GUIDANCE SYSTEM	SHALL	provide the ability	to the Ground Control Station	to switch	between Teleoperation Mode and Mission Mode	
3HL			THE GUIDANCE SYSTEM	SHALL			have	Different Levels of Autonomy	
3.1LL			THE GUIDANCE SYSTEM	SHALL	be able to		operate in	Mission Mode	
3.2LL			THE GUIDANCE SYSTEM	SHALL	be able to		accept	Direct Commands	in Teleoperation Mode
3.3LL			THE GUIDANCE SYSTEM	SHALL	be able to		operate in	Human Pilot Mode	

Table 6.3: Generic Requirements table

6.3 Illustration of Generic Guidance FSM

As shown in the Figure 6.4 the generic UAV guidance system is designed that can be mapped to the ALFURS framework. The graphical representation is done using yED Graph editor. As shown in the table 4.1 and 4.2, our FSM is designed such that it can operate up to ALFURS autonomy level 6. The mapping of autonomy levels is shown using the colour coding of the edges and nodes. Also the ALFUS teleoperation mode is mapped to our FSM. The terms used in naming the edges and states are taking from ALFURS framework or ALFUS framework to keep the generic FSM in close contrast to the generic terms. The FSM operates into three different modes i.e. Human Operated Mode (Remote Control Mode), Teleoperation Mode (Command Mode) and Mission Mode. The mission manager has the options to plan the missions online or offline. In the online planning mode the behaviour sequence is generated in real-time with the vision services. Whereas in the offline mode it is necessary to load data such as 3D World model to plan the mission.

6.4 Summary

The main idea behind using generic agent model is, instead of designing a new model from scratch every time we can make use of the generic model. The generality of a generic model can be considered using two senses i.e. one with respect to the processes or tasks and other with respect to the knowledge structures [19]. The idea of using generality with respect to the process or task is used in designing the abstract and detailed models. A specific model is a model with more specific processes at low level of process abstraction. The agent model explains several process in defining processes within an agent and its interface transitions. This use of generic modelling approach in designing specific cases provides many benefits such as the effort on requirement analysis is saved extensively. Also the requirements can be reused from generic to specific cases. This additionally provides ease of designing specific models because of pre-known structures of the generic case. Also the book by Clements [2] on Software Product Lines describes the benefits of reusing the software product in terms of organisational and timely benefits.

The generic finite state machine is designed according to the ALFURS framework. The ALFURS framework provides reliable source of information over the research on autonomous behaviour of UAS to the aerospace community. The missing gaps in modelling the graphical model and the requirement extraction is taken from the STANAG and ALFUS framework. A set of sequence is followed in the process of model checking the generic model i.e after the requirement analysis these requirements are formalised using the LTL and CTL temporal logics. A graphical model is created keeping in consideration of the system requirements.

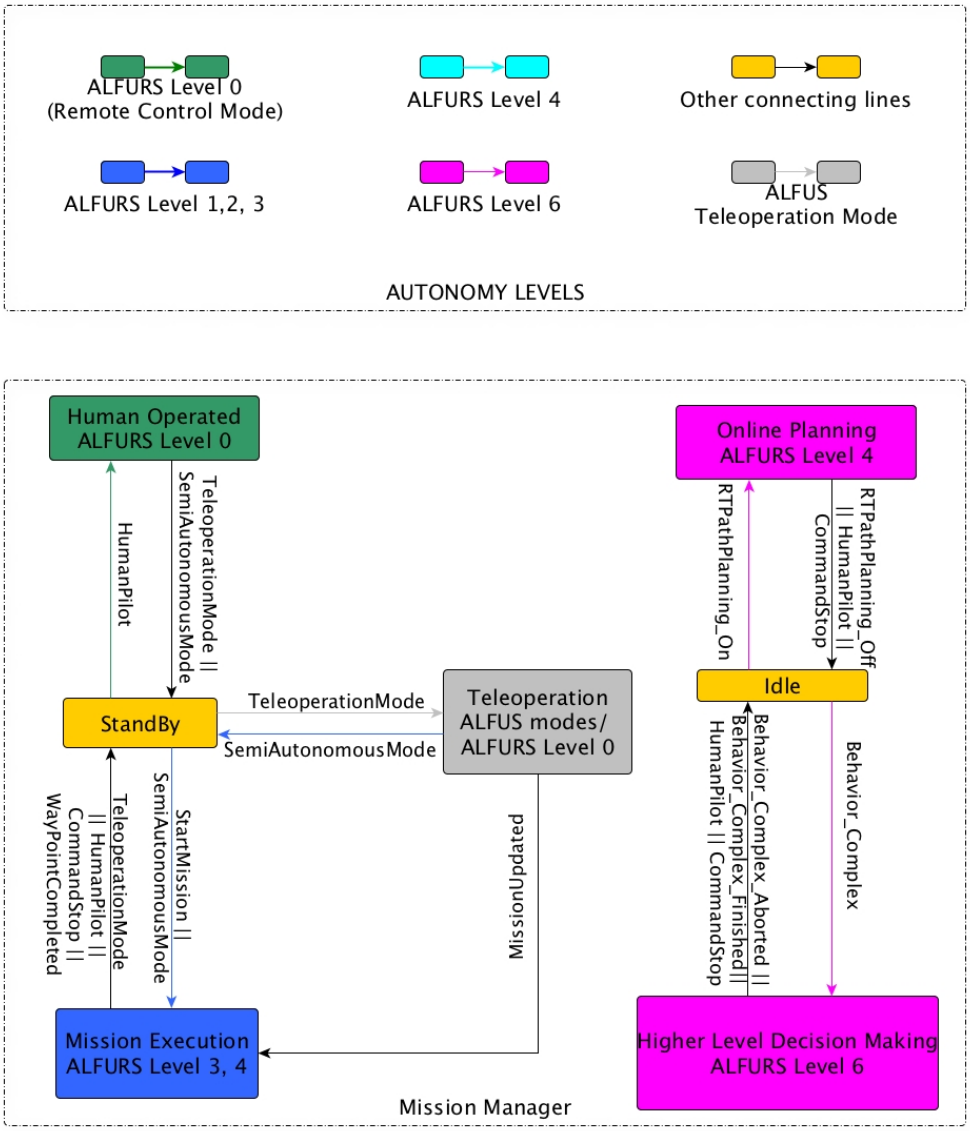

Figure 6.4: Generic Mission Manager FSM Model

This graphical model provides the system behaviour in terms of switching of states as a finite state machine. Finally a NuSMV code is programmed to depict the system behaviour and to translate the graphical model into a formal model. This formal model is verified against the specifications. This gives us a set of requirements that satisfy the system behaviour and the other set of requirements that fails to match the specifications to system behaviour. A change or adaption or refinement in the requirements or graphical model or formalised requirements is necessary when the requirement does not match the system behaviour. Lastly, these satisfied requirements are made to generate counter examples using the trap properties. With these counter examples an approach towards the test generation is enabled as shown in Chapter 8.

Chapter 7

Specific UAV FSM of Mission Manager System

This Chapter provides a detailed overview of model-checking the ARTIS guidance system. Process steps remains to be the same as compared to the model checking of the generic UAV guidance system. Although the source of extraction of requirements and detailed low level information is taken from specific sources unlike a generic framework like ALFURS.

The process steps is as given below:

- Requirement Analysis from specific sources
- Formalising Requirements
- Transition extraction from source code
- Designing of specific FSM
- Formal Modelling using NuSMV model checker
- Counterexample generation using trap properties

The specific UAV in our case is considered as the ARTIS UAV. As mentioned above it can operate upto ALFURS level 6. It can operate in online and offline mission planning modes. The specific model can be mapped directly to the ALFURS autonomy levels which enables us to map the specific model directly to the generic behaviour model. As mentioned in Chapter 4 it has 3 operating modes namely Mission Mode, Command Mode and Remote Control Mode. Our focus is mainly towards designing the FSM of the guidance system of ARTIS. The main component of guidance system is MiPlEx (Mission Planning and Execution) which consists of the Mission Manager. The mission manager provides

replacement to the human guidance system through software. This makes the UAV safety critical as it is more dependent on software.

The Mission manager is capable to take inputs from operator located at the ground control station, vision services during the online mission planning and also works along with the health management system. It provides output to the flight controller and direct the UAV's path. The mission manager consists of the High level and low level behaviours that are predefined motions and missions. The behaviours are organised in the form of a 3T architecture as shown in Figure 4.9. Two major subsystem of the mission manager are the sequence controller and the supervisor. The supervisor consists of the high level behaviours and can control the UAV's operation in online planning mode.

7.1 Specific Requirements Elicitation and Formalization

Similar to the generic requirements, even the specific requirements are formulated using the template based requirements method. Reliable sources were considered while extracting these requirements. Firstly, some of the requirements were gathered from the developers (Christoph Torens and Florian Adolf) through several brainstorming sessions which led to good understanding of the entire ARTIS platform. Secondly, to extract the low level requirements the source code was considered. The source code is implemented in C++ language. Amongst the entire source code, the focus was to extract requirements from "Supervisor.cpp" and "SequenceController.cpp" as these classes implement major parts of focus within the scope of thesis.

The template based requirements method is used to formulate requirements. The table 7.1 shows some of the examples of requirements that were considered during the work. If we compare Table 6.3 and 7.1 we could notice that some of the requirements are same. This denotes that the specific requirements could be reused from the generic requirements.

The formalisation of these requirements is done similar to the formalisation of generic requirements i.e. with the LTL and CTL temporal logics. For example considering the Requirements number 1HL(High Level) from Table 7.1. This is formalised using CTL logic as given below:

SPEC AG EF(SC_State = SlowDown);

Similarly formalisation of Requirement number 1.8LL (Low Level) from Table 7.1 is done using LTL logic as shown below:

LTLSPEC G (SC_State = FlyBuffer eBehaviorAborted) -> X(SC_State = SlowDown);

7.2 Transition extraction from Source Code

The extraction of transitions plays a vital role in designing the specific FSM. Specific behaviour of the UAV is extracted if the FSM is in close relation to the source code. These transitions gives the information about "Start State", " Triggering Condition" and "Next State". The triggering condition defines the transition of one state to another. A total of "112" transitions were extracted from the source code. Some of the transitions are provided in Table 7.2. It shows one of the transitions from "ParseCommand". Similarly other transitions were plotted.

7.3 Mapping of different Artefacts

Mapping or traceability is very essential to track different artefacts. All the requirements are given a "Requirement ID" i.e. a low level requirement with "LL" preceded by a number and similarly high level requirement with "HL" preceded by a number as shown in Table 6.3 and 7.1. These requirement id's are mapped to the formal NuSMV code as well as the graphical FSM. This is helpful in tracking a requirement when the requirement does not satisfy against the system behaviour in NuSMV model checker.

Similarly, the edges in the graphical model is given a unique number that is mapped to the transitions as well as the formal code. This gives advantage in tracking the edge coverage during model checking and also to track the transitions from a counterexample. Any unsatisfied requirement can be easily tracked and corrected depending on the corrective measures to the design or the requirement itself.

7.4 Graphical Illustrations

The model of the system need to be very precise according to the actual system because any verification using model-based techniques is only as good as the

Req. ID	Logical/ Temporal Conditions	Precondition	System under focus	Legal Relevance	Activity	Whom?	What?	Process Word	Additional Details
1HL	Always		THE GUIDANCE SYSTEM	SHALL	be able to		to goto	SlowDown state	or another safe state
1.1LL	On CommandStop during mission		THE GUIDANCE SYSTEM	SHALL	be able to		to goto	SlowDown state	
1.2LL	If arDirectGCS Command is executed		THE GUIDANCE SYSTEM	SHALL	be able to		to goto	SlowDown state	
1.3LL	During timeout in arDirectGCS command		THE GUIDANCE SYSTEM	SHALL	be able to		to goto	SlowDown state	
1.4LL	If manual pilot takes control		THE GUIDANCE SYSTEM	SHALL	be able to		to goto	SlowDown state	
1.5LL	Always		THE GUIDANCE SYSTEM	SHALL	have/ implement			Safe Flight state	
1.6LL	Always		THE GUIDANCE SYSTEM	SHALL	have/ implement			Safe Ground state	
1.7LL	In an aborted or unfinished		THE GUIDANCE SYSTEM	SHALL	allow to		to goto	SlowDown state	
1.8LL	Always		THE GUIDANCE SYSTEM	SHALL	allow to		to goto	SlowDown state	for all behaviours
2HL			THE GUIDANCE SYSTEM	SHALL			have	Different Levels of Autonomy	
2.1LL			THE GUIDANCE SYSTEM	SHALL	allow to		operate in	Mission Mode	
2.2LL			THE GUIDANCE SYSTEM	SHALL	be able to		accept	Direct Commands	in Teleoperation Mode
2.3LL			THE GUIDANCE SYSTEM	SHALL	be able to		operate in	Human Pilot Mode	

Table 7.1: Specific Requirements table

Start State	Triggering Condition	Next State
ParseCommand	CManualPilot	MissionControllerOff
ParseCommand	CCommandStop or !bMissionMode	Slowdown
ParseCommand	CMissionFinished	Slowdown
ParseCommand	CDirectCommandValid_GCS_Vel	VelGCS
ParseCommand	CGroundHeightAvailable and eBehaivor_Basic_Landing	Landing
ParseCommand	eBehavior_Basic_TakeOff	TakeOff
ParseCommand	eBehaivor_Basic_WaitingFor	WaitingFor
ParseCommand	eBehavior_Basic_HoverTo	HoverTo
ParseCommand	eBehavior_Basic_FlyTo	FlyTo
ParseCommand	eBehavior_Basic_FlyBuffer	FlyBuffer
ParseCommand	eBehavior_Basic_CircleAroundXY	CircleAroundXY
ParseCommand	eBehavior_Basic_PiroutteFlightXYZ	PiroutteFlightXYZ
ParseCommand	eBehavior_Basic_HammerHeadTurn	HammerHeadTurn
ParseCommand	eBehavior_Basic_StandBy	Slowdown
ParseCommand	eBehavior_Basic_Slowdown	Slowdown
ParseCommand	NULL cmd	Slowdown
ParseCommand	GUARD_SELF	ParseCommand

Table 7.2: Transition of States from source code

model of the system. The pictorial representation of the finite state machine has to be created before describing the model in NuSMV. The graphical representation is done using yED Graph editor. Our model is a modular design that works independent of each other and is capable of representing the Sequence Controller and Supervisor with a clear distinction. The transitions are triggered by changes in the state. The transitions are drawn with different colour and manifestation to map to the ALFURS autonomy levels. Every colour signifies to different ALFURS autonomy levels. Edges are marked with different numbering so that we can track every requirement to its specification. This allows us to have a link between requirements list, NuSMV code, formal specification and the design itself.

7.4.1 Abstract MM Model

The abstract models of Sequence Controller and Supervisor is as shown in Figures 7.1 and 7.2 respectively. Both the FSM are created using abstraction of the detailed models (Figures 7.3 and 7.4). The sequence controller and supervisor finite state machines are coded into one NuSMV code. Both the FSM function independently although the supervisor controls the operation of the sequence controller.

The abstract model complies to all the high level ARTIS specific requirements. The process of abstraction is done on the detailed models Figures 7.3 and 7.4). For example all the low level behaviours in detailed model is abstracted into one state i.e. “Predefined Behaviours”. Also the presence of “VelGCS” and “TrajectoryCommands” in the detailed model is abstracted into a single state “man_trans_command” (manual transmission command).

Similarly the “StopAndGo” functionality in detailed model is abstracted into “Main/ Parser” in the abstract model. The abstract model can also be mapped directly to the ALFURS autonomy levels.

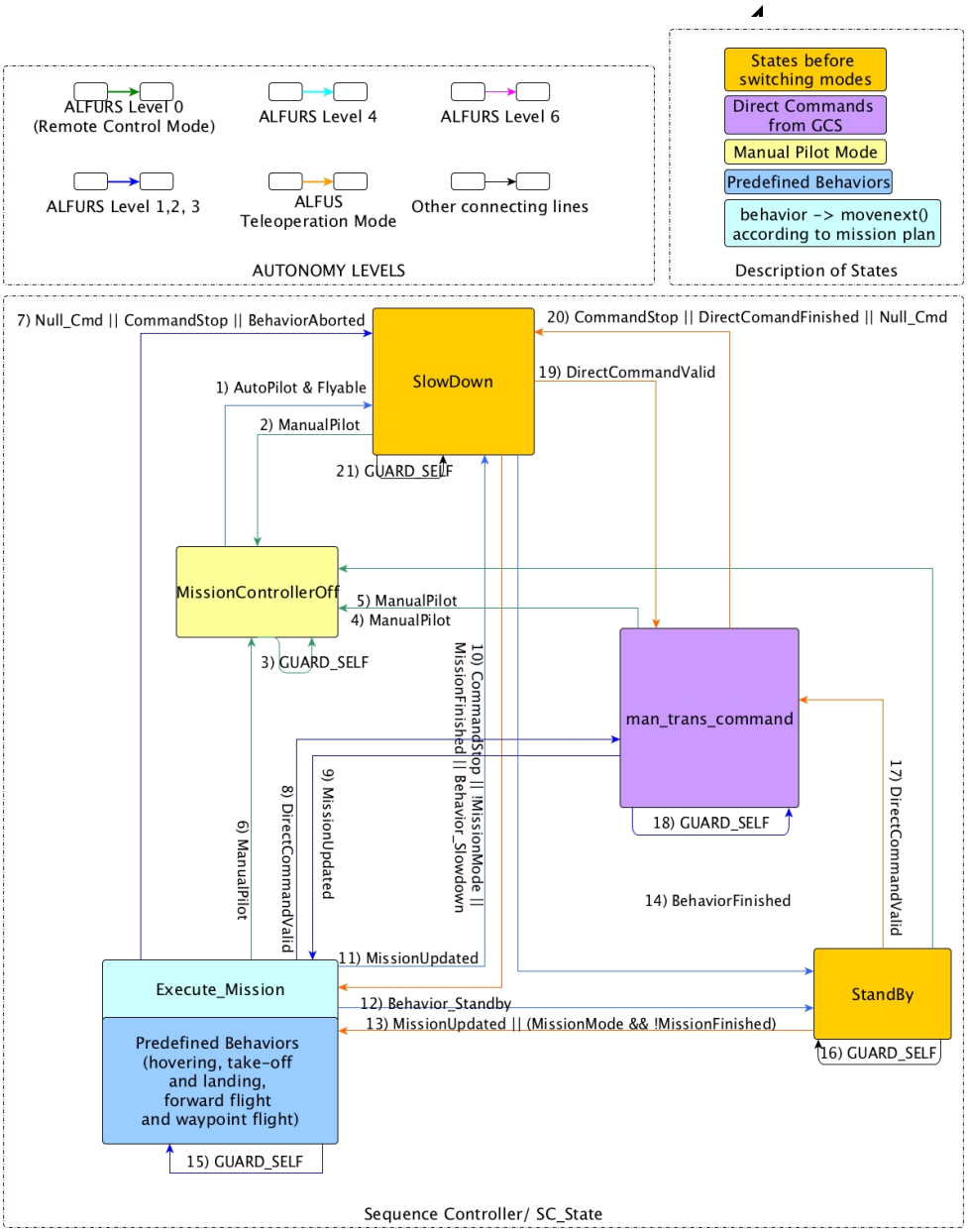

Figure 7.1: Specific Mission Manager (Sequence Controller) abstract FSM Model

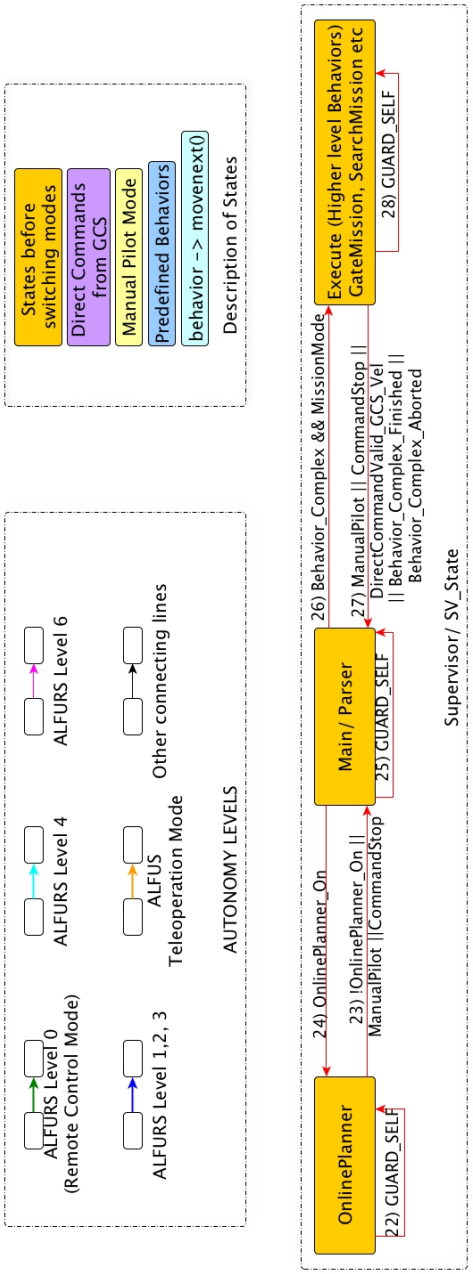

Figure 7.2: Specific Mission Manager (Supervisor) abstract FSM Model

7.4.2 Detailed MM Model

The detailed model complies to the low level requirements. It is established in close relevance to the source code. The "MissionControllerOff" state depicts the Remote Control operation as the mission manager is turned off during this operation as the guidance of UAV is solely done by the human pilot. Whenever a "CManualPilot" command is triggered all the states switches to the "MissionControllerOff" state. The "SlowDown" depicts the smooth transfer of different modes of operation i.e. either before switching from Manual Pilot to Command Mode or from Mission Mode to the Command Mode. The "VelGCS" state (Velocity command from ground control station) and "TrajectoryCommand" represents the mission manager operation in Command Mode. The "StandBy" state is used whenever a behaviour is finished and the mission manager is waiting for next consequent commands.

The Supervisor operates on a higher level and monitors or influences the control on the Sequence Controller. The Supervisor is mainly responsible to handle higher level commands like *"Gate Mission"* as shown in Figure 7.4 whereas the Sequence Controller handles low level behaviours such as *"HoverTurn"* or *"Take-Off"* as shown in Figure 7.3 . Some of the low level behaviours are present in the FSM that are specific to ARTIS UAV. It contains behaviours to wait at a position (WaitFor), turn on the spot (HoverTurn), fly along a linear trajectory towards a location (HoverTo), to fly around a point along a horizontal circular trajectory (Pirouette), and to do fast forward flight along an arbitrary trajectory (FlyTo).

7.5 Specific Model Checking

As discussed in chapter 3. Formal Methods offers early integration of verification in the design process and makes the verification more efficient which in-turn reduces the verification time. Model-based verification techniques are based on models describing the possible system behaviour in a mathematical manner. The main advantage of model checking is that before any verification, it leads to the discovery of incompleteness and inconsistencies in informal specifications. The system models systematically explores all the states of the system model.
Model checking explores all the possible system states available to trace. It examines all possible system scenarios in a systematic manner. This means that the errors that cannot be discovered by review and testing as mentioned in the above section can be found using the model checkers.

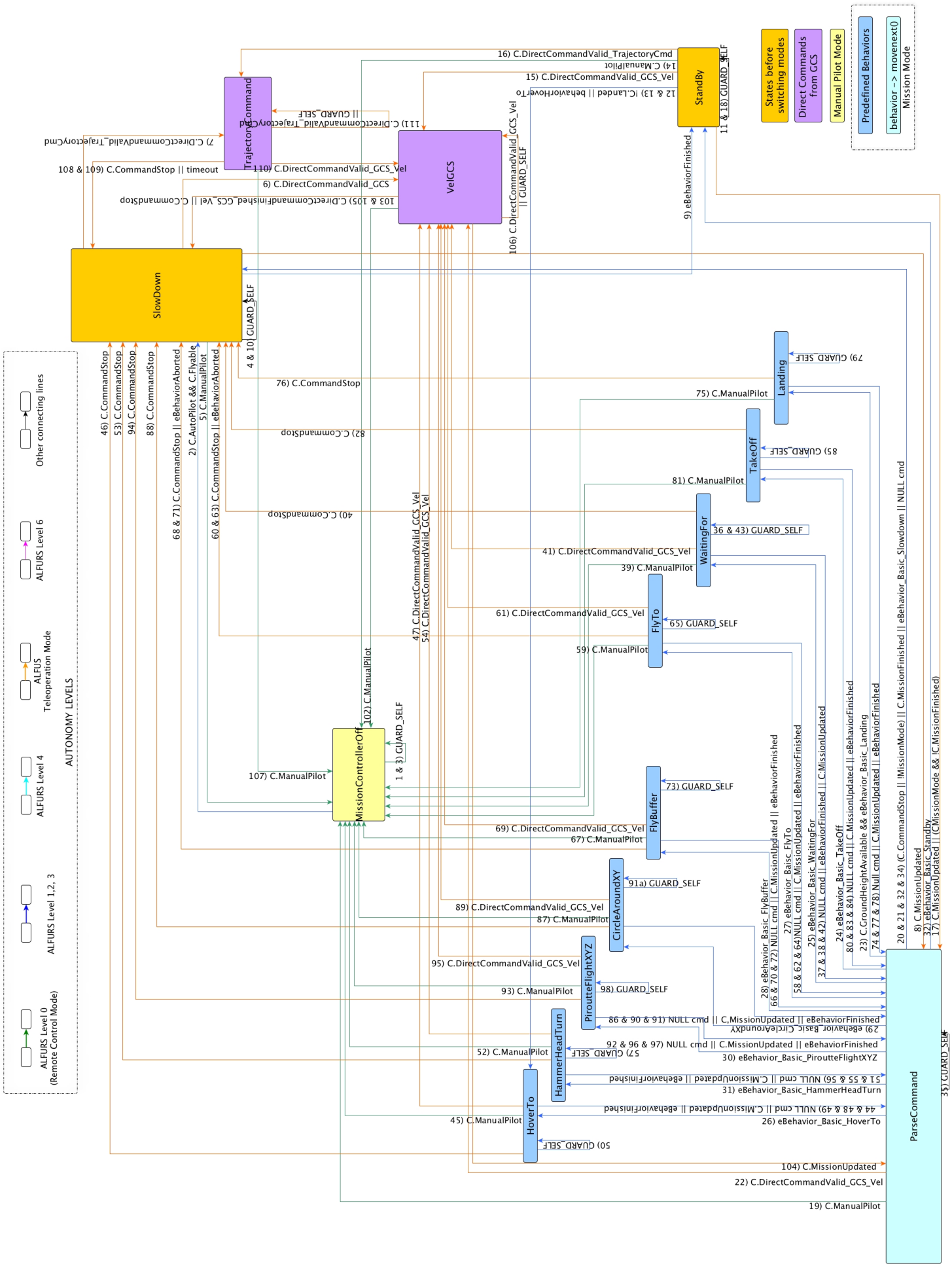

Figure 7.3: Specific Mission Manager (Sequence Controller) Detailed FSM Model

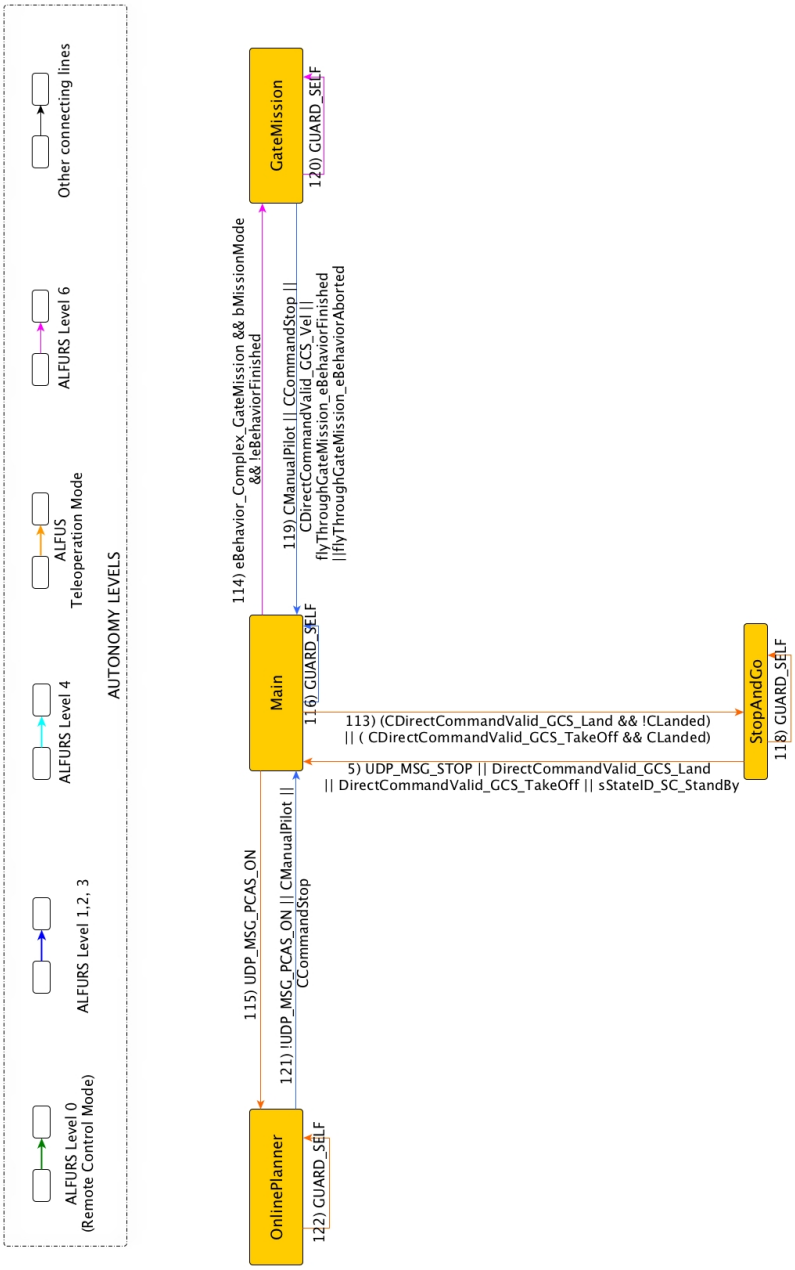

Figure 7.4: Specific Mission Manager (Supervisor) Detailed FSM Model

7.6 Summary

A detailed overview of model checking of ARTIS guidance system is presented. The process involves requirement analysis, these requirements are extracted from specific sources like source code, documentation, meeting with software developers. Later these semi formal requirements written in template based method are converted to mathematical form using temporal logics, A important source of information to design specific finite state machine is to know when the state transition is done by which triggering command or condition. Hence the transitions are extracted from the source code "SequenceController.cpp" and "Supervisor" . These transition need to connected to each other in the form of graphical model to obtain the finite state machine.

Later the model checking is done by formal modelling the graphical model into NuSMV. The model is verified against the the specifications. Any error in the model or requirement is encountered by a property that does not satisfy the model. The final step is to generate counterexamples using the trap properties by using equivalence transformation. This gives a finite set of states that show why the property is not violated.

Chapter 8

Test Case Generation

Verification and Validation (V & V) consumes approximately 50% to 70% of software development resources [25] for high assurance system or safety critical systems. Even though model checking provides greater advantage compared to testing, testing still remains to be a vital part of V & V technique. In current trends majority of time is denoted to development of test cases to test the necessary functionality of software against the specified behaviour [25]. Thus if the process of deriving test cases could be automated than most of the time and cost is saved in the process of testing a software. Model checkers can be used to automatically generate test sequences. Such that these test sequences in turn aid for a predefined structural coverage of a formal specification.

In this chapter, it is shown how the formal methods can be used to generate structural tests from a formal specification of the respective system behaviour. One of the major benefit of using NuSMV is its capability of generating counterexample when the specification does not satisfy the system behaviour. This is because model checkers explores the reachable state space searching for violations of the properties under investigation. The counterexample explains how the violation can take place in terms of finite state transition from the initial state of the finite state system to the position within the finite state transition where the violation occurs.

8.1 Counterexample Generation using Trap Properties

As explained in the above section, the counterexamples provides a set of transition within the finite state machine and depicts the position where the violation occurs. But this is true when the specification does not hold “TRUE” against the

system behaviour. But to find the test sequences it is necessary to purposefully generate a counterexample for the specifications that are "TRUE" otherwise. For example, we can force the model checker that a certain transition in a specification does not transit. This would challenge the model checker to generate a sequence of inputs with respective outputs to display that this transition actually exists. This provides us our necessary approach towards test case generation as a test sequence is automatically generated by the model checker.

This assertion of transition is possible using the trap properties as mentioned in [4]. The trap property violates the satisfying property of the specification and forces the model checker to generate a counter example. The counterexample assigns a sequence of values to the inputs and outputs of the system making the counterexample as a test sequence.

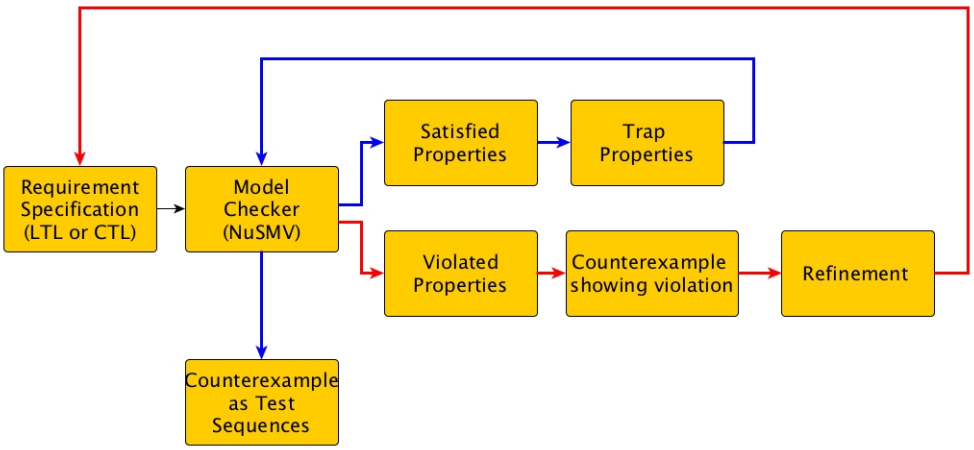

Figure 8.1: Process of negating a satisfied property

As shown in Figure 8.1, the process of generating the counterexamples as test sequences is depicted. It is necessary to use the trap properties only to the specifications that satisfy against the system behaviour. Or in other words, the system has to be model checked and violating specifications needs to be refined before applying the trap properties. As shown in the figure 8.1 the process flow according to the *red edges* also shows that a counterexample is generated during the process but this is not our required test sequence. As the counterexample in this case shows the violation of specification to the system behaviour. Our required test sequence is generated when the *blue edges* are followed i.e. an equivalence transformation is applied to the specification to generate these counterexamples.

One way would be to use equivalence transformation. The LTL logics are explained in Chapter 2. Negation propagation for LTL as shown below:

$$\neg X\Phi \equiv X\neg\Phi$$

$$\neg G\Phi \equiv F\neg\Phi$$

$$\neg F\Phi \equiv G\neg\Phi$$

Although we could achieve a negation of LTL and CTL properties by using a simple "exclamatory **!**" sign because the NuSMV tool is capable of this easy form of negation but the position of application of the symbol is important to ensure its correctness. For example considering a requirement 1.8 LL from Table 7.1: "*Always the system should allow to goto Standby after all flight behaviours*" the LTL specification would be:

$$LTLSPEC\ G\ (SCM\_State = TakeOff)\ ->\ F(SCM\_State = Standby);\\ --Specification\_1a$$

In the above specification "SCM_State" refers to Sequence Controller Mission States and "Takeoff" and "Standby" are the behaviours in the mission scenario explained in next section according to figure 8.3. The requirement mention "Always" that means that this specification should be Global to entire system and hence "G". After application of the negation property the above specification appears as follows:

$$LTLSPEC\ !G\ ((SCM\_State = TakeOff)\ ->\ F(SCM\_State = Standby));\\ --Specification\_1b$$

Similarly, considering another requirement for a CTL specification: "*The system shall be able to Takeoff*" the CTL specification appears to be as follows:

$$SPEC\ AG\ EF(SCM\_State = Takeoff); --Specification\_2a$$

After applying the negation property the specification is as follows:

$$SPEC\ AG\ !EF(SCM\_State = Takeoff); --Specification\_2b$$

8.2 Test Case Scenario

A mission plan (highest-level task assigned to a unmanned system according to ALFUS) or behaviour sequence assigned by operator to UAV is considered as

a test scenario to explain the generation of test sequences. The test mission scenario is taken from [14] is as shown in Figure 8.2 on the left. The defined mission is a sequence of behaviour commands for traversing into appropriate states. The system cannot reason about the sensibility of the mission plan but the ENBF grammar (as shown in Figure 8.2 on the right) checks for the plausibility of the mission plan [14]. It ignores malformed behaviour sequences that does not match to its grammar. For example the ENBF enforces every mission to start with a "TakeOff" behaviour and finish with a "Landing" behaviour. The Figures

```
ID 20070510
TO -5
HV 0 0 -3 180
avoidance on
WT 10
HV 33.3 3.51 -7 28.9
HV 37.1415 1.63484 -10 90
avoidance off
WT 5
tracker on
HV 37.1415 48.688 -10 90
HV 37.1415 48.688 -10 180
HV 31.81 48.688 -10 180
HV 26.4785 1.63484 -10 180
HV 26.4785 1.63484 -10 90
HV 26.4785 48.688 -10 90
HV 26.4785 48.688 -10 180
HV 21.147 48.688 -10 180
tracker off
LD
WO
```

```
<mission>     ::= "ID" <numbers> <takeoff> [<waypointlist>] <land>;
<numbers_all> ::= {<nonzero> | "0"};
<nonzero>     ::= "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9";
<float>       ::= [ [<numbers_all>] "." ] <numbers_all>;
<numbers>     ::= <nonzero> {<nonzero> | "0"};
<waypointlist>::= { <reactive> | <trajectory> | <task> };
<reactive>    ::= {<takeoff> | <land> | <standby >};
<trajectory>  ::= {<hover> | <fastflight> | <pir> | <pir_flight>};
<takeoff>     ::= "TO" [ "-" <float> ];
<hover>       ::= <hover_to> | <hover_turn> | <hover_wait>;
<hover_to>    ::= "HV" ["-"]<float> ["-"]<float> ["-"]<float> ["-"]<float>;
<hover_turn>  ::= "HT" ["-"]<float> ["-"]<float>;
<hover_wait>  ::= "WT" [<number>];
<standby>     ::= "WO";
<fastflight>  ::= <fly_to> <fly_to> <fly_to> { <fly_to> };
<fly_to>      ::= "FT" ["-"]<float> ["-"]<float> ["-"]<float> ["-"]<float>;
<pir>         ::= "PI" ["-"]<float> ["-"]<float> ["-"]<float> ["-"]<float>;
<pir_flight>  ::= "PF" ["-"]<float> ["-"]<float> ["-"]<float> ["-"]<float>;
<task>        ::= <command_src> "on" <waypointlist> <command_src> "off";
<command_src> ::= "tracker" | "avoidance";
<land>        ::= "LD";
```

Figure 8.2: Mission Scenario (left) and ENBF grammar (right) [14]

7.3 and 7.4 has modified and only the behaviours within the mission plan is considered as shown in Figure 8.3 to consider a test scenario. As the mission plan is executed in the mission mode, all other modes such as the remote pilot mode and the command mode is not considered in this example. Thus the "SlowDown" state is eliminated as there is no change over in different modes of operation. The "ParseCommand" is is used as for each behaviour there exists a termination condition or to have transition between different behaviours considered in the mission plan. The "ParseCommand" is an important state to execute the next behaviour within the mission plan. In the mission plan shown in Figure 8.2 the UAV takes-off (TakeOff State) and hovers at the position followed by hovering to a specified location and makes a hover turn (HammerHeadTurn state in our case) then waits at the location (WaitingFor state) followed by a fast flight to a location and performs its task and land at that position. This scenario can be covered in our FSM. The ENBF grammar ensures that a "HoverTo" command is not executed after a "Landing". But this functionality can be adopted during the description of states in the NuSMV model checker.

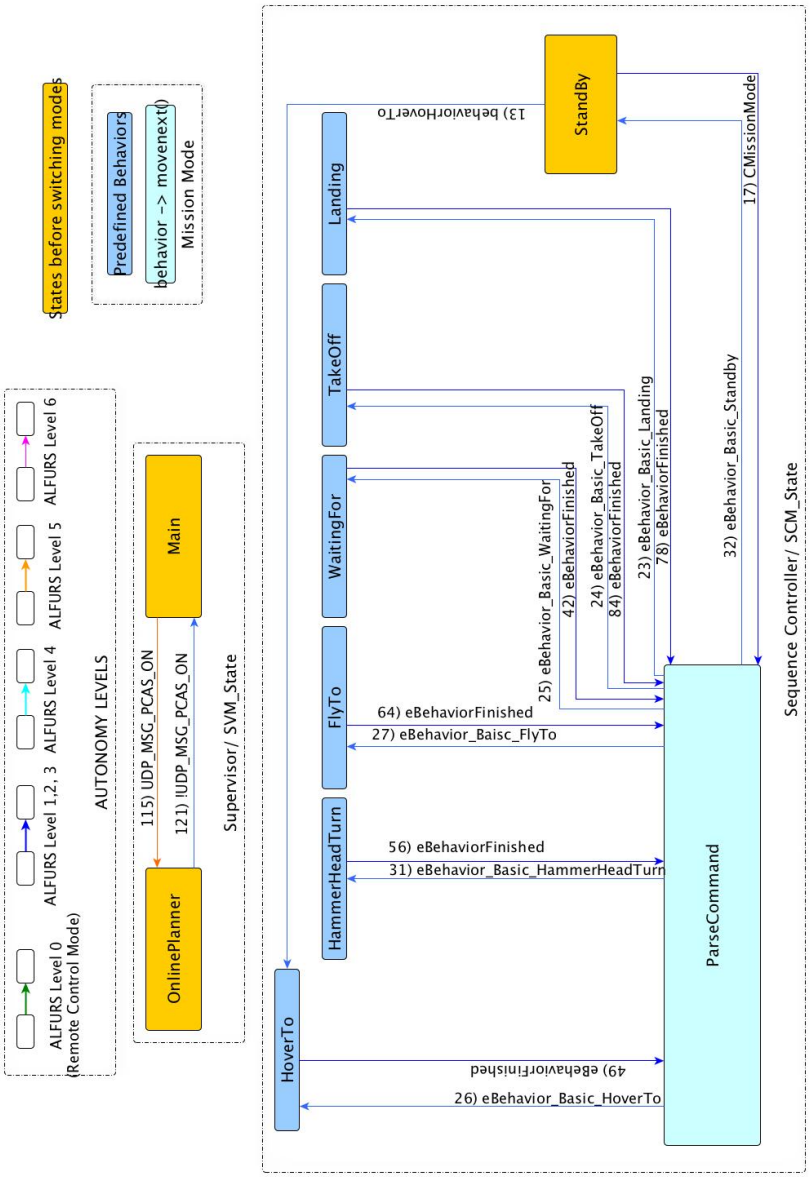

Figure 8.3: Mission Scenario from Specific Mission Manager Detailed FSM Model

The mission scenario of graphical model is formal modelled in NuSMV and model checking is performed with the appropriate requirements suiting to the mission plan. Even in this case the modules can be reused from the formal model of specific guidance system and a similar methodology is followed for model checking.

```
-- specification AG !(EF SCM_State = TakeOff)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
 -> State: 1.1 <-
   SVM_State = Main
   SCM_State = TakeOff
   Command4Mission = eBehavior_Basic_TakeOff
   UDP_MSG_PCAS_ON = FALSE
   eBehaviorFinished = FALSE
```

Figure 8.4: CTL Counterexample

As explained in the previous section, the trap properties are used to force the model checker to generate counterexamples. The generated counterexamples are as shown in the next section.

8.3 Understanding Counterexamples for Test Cases

The counterexample generation is a very powerful functionality in understanding the violation of any specification against its system behaviour. Also there is another advantage as it also serves as a test sequence when used along with trap properties to the specifications. It shows a finite state of transition from the initial state of a system up to the state where the violation occurs. For example considering Specification_1b and Specification_2b the LTL and CTL counterexample are generated. The LTL counterexample of Specification_1b is as shown in Figure 8.5 and the CLT counterexample of Specification_2b is as shown in Figure 8.4.

The requirement of Specification_2a mentions that “The system shall be able to TakeOff”. When a trap property is applied Specification_2b is formulated. According to Specification_2b it means that the system shall not be able to goto TakeOff state, which is not true and hence a counterexample is generated showing that the system traces the “TakeOff” state right in the initial State 1.1 according to Figure 8.4. Similarly when we consider the requirement of Specification_1a which states “Always the system should allow to goto Standby after all flight behaviours”. After applying trap properties this means that after the flight behaviours (TakeOff is considered in this case) the system cannot transit to the “StandBy State” but this is not possible according to the system behaviour and hence a counterexample is generated. As seen in the Figure 8.5 the system starts from its initial SCM_State i.e. TakeOff (marked in figure under State: 2.1), it

traces different path and finally reaches StandBy state (marked in figure under State: 2.31). This gives us a test sequence of transition through different paths. And it finally loops back to its start state i.e. SCM_State = TakeOff (marked in figure under State:2.34)

8.4 Summary

A test case is a set of conditions under which a tester will determine whether a software system and it behaviour or features is working without any defects as it was planned to be established. The test cases can ensure parts of structural coverage of a model. In this work our two test case are assigned to each requirements i.e. a positive and a negative test case or a satisfied property and a counterexample generated sequence. The NuSMV provides a counterexample for a negated property and forces the inputs and display the output in terms of finite trace of states. In other words the current state as a precondition and the next transited state or output state as a post condition. The commands trigger the switching of states. Although testing is not replaceable completely using the process of counterexample generation as test sequences but it provides a benefit of automation in generating the test sequences. An instantiation process needs to be carried out to these counterexamples to have a test generation.

A mission plan is considered as a test scenario. The mission plan consists of a behaviour sequence with several flight behaviours. All the flight behaviours and consequent states are covered in the elicitated requirements. This provides us a specification to be tested for all the edges present in the finite state machine. When a counterexample is generated it traverses through different states and depicts that all the edges of the model are considered in the test sequences.

The process of using trap properties helps in generating the counterexamples. These trap properties negate the satisfied property and thus forces the NuSMV model checker to generate test sequences. This work gives an approach towards the test generation using model checking.

specification !(G (SCM_State = TakeOff -> F SCM_State = StandBy)) is false

```
1   – as demonstrated by the following execution
2   sequence
3   Trace Description: LTL Counterexample
4   Trace Type: Counterexample
5    -- Loop starts here
6    -> State: 2.1 <-
7      SVM_State = Main
8      SCM_State = TakeOff
9      Command4Mission =
10  eBehavior_Basic_TakeOff
11     UDP_MSG_PCAS_ON = FALSE
12     eBehaviorFinished = FALSE
13   -> State: 2.2 <-
14     Command4Mission =
15  eBehavior_Basic_Landing
16   -- Loop starts here
17   -> State: 2.3 <-
18     Command4Mission =
19  eBehavior_Basic_TakeOff
20   -> State: 2.4 <-
21     Command4Mission =
22  eBehavior_Basic_HoverTo
23   -> State: 2.5 <-
24     Command4Mission = eBehavior_Basic_FlyTo
25   -> State: 2.6 <-
26     Command4Mission =
27  eBehavior_Basic_HammerHeadTurn
28   -> State: 2.7 <-
29     Command4Mission =
30  eBehavior_Basic_WaitingFor
31   -> State: 2.8 <-
32     Command4Mission =
33  eBehavior_Basic_StandBy
34   -> State: 2.9 <-
35     Command4Mission =
36  eBehavior_Basic_TakeOff
37     eBehaviorFinished = TRUE
38   -> State: 2.10 <-
39     SCM_State = ParseCommand
40     eBehaviorFinished = FALSE
41   -> State: 2.11 <-
42     SCM_State = TakeOff
43     eBehaviorFinished = TRUE
44   -> State: 2.12 <-
45     SCM_State = ParseCommand
46     Command4Mission =
47  eBehavior_Basic_StandBy
48     eBehaviorFinished = FALSE
49   -> State: 2.13 <-
50     SCM_State = StandBy
51     Command4Mission =
52  eBehavior_Basic_TakeOff
53   -> State: 2.14 <-
54     Command4Mission = CMissionMode
55   -> State: 2.15 <-
56     SCM_State = ParseCommand
57     Command4Mission =
58  eBehavior_Basic_HoverTo
59   -> State: 2.16 <-
60     SCM_State = HoverTo
61     Command4Mission =
62  eBehavior_Basic_TakeOff
63   -> State: 2.17 <-
64     eBehaviorFinished = TRUE
65   -> State: 2.18 <-
66     SCM_State = ParseCommand
67     Command4Mission = eBehavior_Basic_FlyTo
68     eBehaviorFinished = FALSE
69   -> State: 2.19 <-
70     SCM_State = FlyTo
71     Command4Mission =
72  eBehavior_Basic_TakeOff
73   -> State: 2.20 <-
74     eBehaviorFinished = TRUE
75   -> State: 2.21 <-
76     SCM_State = ParseCommand
77     eBehaviorFinished = FALSE
78   -- Loop starts here
79   -> State: 2.22 <-
80     SCM_State = TakeOff
81   -> State: 2.23 <-
82     eBehaviorFinished = TRUE
83   -> State: 2.24 <-
84     SCM_State = ParseCommand
85     Command4Mission =
86  eBehavior_Basic_Landing
87     eBehaviorFinished = FALSE
88   -> State: 2.25 <-
89     SCM_State = Landing
90     Command4Mission =
91  eBehavior_Basic_TakeOff
92   -> State: 2.26 <-
93     eBehaviorFinished = TRUE
94   -> State: 2.27 <-
95     SCM_State = ParseCommand
96     Command4Mission =
97  eBehavior_Basic_HammerHeadTurn
98     eBehaviorFinished = FALSE
99   -> State: 2.28 <-
100    SCM_State = HammerHeadTurn
101    Command4Mission =
102 eBehavior_Basic_TakeOff
103  -> State: 2.29 <-
104    eBehaviorFinished = TRUE
105  -> State: 2.30 <-
106    SCM_State = ParseCommand
107    Command4Mission =
108 eBehavior_Basic_StandBy
109    eBehaviorFinished = FALSE
110  -> State: 2.31 <-
111    SCM_State = StandBy
112    Command4Mission =
113 eBehavior_Basic_TakeOff
114  -> State: 2.32 <-
115    Command4Mission = CMissionMode
116  -> State: 2.33 <-
117    SCM_State = ParseCommand
118    Command4Mission =
119 eBehavior_Basic_TakeOff
120  -> State: 2.34 <-
121    SCM_State = TakeOff
122
123
124
125
126
127
128
129
130
```

Figure 8.5: LTL Counterexample

Chapter 9

Experimental Results

The overall results gathered during this thesis is briefed in this Chapter. In Chapter 5 gives the concept of the thesis. Chapters 6, 7 and 8 provides detailed information about the contribution to this thesis and the results achieved. Chapter 6 explains the implementation of generic modelling approach by mapping the information flow model of GNC component of UAV to the GAM. Chapter 6 also shows the result of extracted generic requirements in Table 6.3. These requirements were formalised using LTL and CTL properties. According to one of the goal of this thesis a generic model (Figure 6.4) is created using the ALFURS framework and according to the research question the specific cases can be derived using the generic approach as shown in Figures 7.3 and 7.4 (showing detailed FSM of Sequence Controller and Supervisor respectively).

Chapter 7 explains about the specific model checking of UAV mission manager system. It provides the sources considered for requirements extraction. Table 7.1 gives a snippet of the extracted requirements. Also section 8.1 gives some examples of formalising these requirements using temporal logics. It explains the process of transition extraction from the code and a part of the transition is given in Table 7.2. Chapter 7 also explains how the mapping of different artefacts is helpful in tracking amongst all the requirements and thus fault finding. The graphical models of abstract sequence controller and supervisor is shown in Figure 7.1 and Figure 7.2 respectively. A more detailed version of the graphical models is shown in Figures 7.3 and 7.4.

A total of 4 FSMs were created and model checked using NuMSV as summarised in the Table 9.1. These models were refined according to the property satisfying the FSM. Chapter 8 gives the result of counterexamples generated using trap properties which can be used as test sequences. 8.5 and 8.4 shows the counterexample generated from applying trap property to a LTL and CTL logics respectively.

Description	Figure Number
Generic Mission Manager FSM Model	6.4
Specific Mission Manager (Sequence Controller) abstract FSM Model	7.1
Specific Mission Manager (Supervisor) abstract FSM Model	7.2
Specific Mission Manager (Sequence Controller) Detailed FSM Model	7.3
Specific Mission Manager (Supervisor) Detailed FSM Model	7.4
Mission Scenario from Specific Mission Manager Detailed FSM Model	8.3

Table 9.1: List of FSMs Designed

Chapter 10

Conclusion and Future Scope

In this work, the research is carried on generic modelling approach so that the above mentioned generic model be extended and specialised to reflect specific cases like ARTIS. This allows the (re)use of these generic models as a template rather than building or designing a model from scratch. This work addressed the design of generic models that could be reflected or abstracted to specific cases.

Furthermore, the reuse of artefact amongst generic and specific cases initiated right from the first step i.e. the requirement extraction as some of the requirements were common and could be reused amongst both the cases. These shared requirements could also be used for other specific UAV's. The use of TL provides a uniform notation for expressing wide range of requirements. These shared requirements also reduces the time in formalising them using temporal logics as the temporal logics can be reused just by relating to the generic or specific terms. As a modelling point of view the generic model could be abstracted from specific cases. This serves as a basis for designing other specific cases similar to the software product line approach.

The objective to research was to use of generic formal models for model-checking and test generation using model-checking. This objective was addressed during this work. But due to lack of low level behaviours of generic models generating tests with specific inputs is challenging . Instead a basis of test generation approach using model-checking is performed over specific models. The availability of low level behaviour in specific UAV allow the user to test the desired outputs for a set of inputs as this characteristics is not available with the generic models due to its abstract nature.

An approach towards test case generation involves generation of test sequences or counterexamples using trap properties on the properties that are TRUE otherwise. These test sequences gives us parts of structural coverage. To ensure better structural coverage a mission scenario was considered from the specific UAV. The Figures 7.3 and 7.4 were was restructured to suit the mission scenario as shown in Figure 8.3. A wide set of requirements helps in providing better structural coverage as the negation of every satisfied property makes the model checker to visit untraced transitions for generation of a counterexample.

One of the limitation could be correctness of the designed model influences directly on the generated tests. A designed FSM should be in close relation to its requirements or real world behaviour. Any missing gaps between the software system and the formal model needs to be bridged by formulating good requirements.

As a future scope, these test sequences or counterexample generated using trap properties can be further harnessed to create executable test cases and followed by test case generation framework. So that the other specific UAVs can benefit from this framework rather than attempting to build a set of test cases from the scratch. A similar approach can be followed for other specific cases.

Bibliography

[1] Henrik Reif Andersen. An introduction to binary decision diagrams. Lecture Notes for Efficient Algorithms and Programs, September 1999.

[2] Paul Clements. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, August 2001.

[3] DLR. Dlr-institute of flight systems www.dlr.de, 2015.

[4] Angelo Gargantini et al. Using model checking to generate tests from requirement specifications. In *Software Engineering - ESEC/FSE*, pages 146–162, Italy, 1999. Springer.

[5] Christel Baier et al. *Principles of Model Checking*, volume 26202649. MIT Press Cambridge, Cambridge, May 2008.

[6] Christoph Torens et al. Automated verification and validation of an onboard mission planning and execution system for uavs. In *AIAA Infotech@Aerospace 2013*, Boston, August 2013. American Institute of Aeronautics and Astronautics, Inc.

[7] Christoph Torens et al. Software verification consideration for the artis unmanned rotorcraft. In *51st AIAA Aerospace Sciences Meeting*, Grapevine,TX,USA, 2013. American Institute of Aeronautics and Astronautics, Inc.

[8] Christoph Torens et al. Formal requirements and model-checking for vv automation of a rpas mission management system. In *AIAA Infotech@Aerospace*, FL USA, January 2015. American Institute of Aeronautics and Astronautics, Inc.

[9] Corina S P et al. Symbolic execution and model checking for testing. In *Hardware and Software: Verfication and Testing*, pages 17–18. Springer, 2008.

[10] Dirk Beyer et al. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering*, pages 326–335, USA, 2004. IEEE Computer Society.

[11] Edmund M Clarke et al. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, April 1986.

[12] Fahroo et al, editor. *Recent Adavances in Research on Unmanned Aerial Vehicles*, volume 444. Springer, Springer-Verlag Berlin Heidelberg, 1 edition, 2013.

[13] Florian-Micheal Adolf et al. Probabilistic roadmaps and ant colony optimization for uav mission planning. In *Intelligent Autonomous Vehicles*, volume 6, pages 264–269, Braunschweig, Germany, 2007. German Aerospace Center (DLR).

[14] Florian-Micheal Adolf et al. A sequene control system for onboard mission management of an unmanned helicopter. In *AIAA Infotech@Aerospace 2007*, Rohnert Park, California, May 2007. AIAA, American Institute of Aeronautics and Astronautics, Inc.

[15] Florian-Micheal Adolf et al. Behaviour-based high level control of a vtol uav. In *AIAA Infotech@Aerospace 2009*, page 13, Washington, April 2009. American Institute of Aeronautics and Astronautics, Inc.

[16] Florian-Micheal Adolf et al. Vision-based target recognition and autonomous flights through obstacle arches with a small uav. In Anibal Ollero, editor, *AHS 65th Annual Forum Proceedings*, pages 259–280, Grapevine,TX,USA, May 2009. AHS International.

[17] Florian-Micheal Adolf et al. Multi-query path planning for exploration tasks with an unmanned rotorcraft. *AIAA Infotech@Aerospace*, 19:20, 2012.

[18] Florian-Micheal Adolf et al. Trajectory time reduction using field of view-based smoothing of roadmap-based paths. In *AHS 69th Annuak Forum Proceedings*. AHS International, Inc, May 2013.

[19] France M T Brazier et al. Compositional design and reuse of a generic agent model. *Applied Artifical Intelligence*, 14(5):491–538, 2000.

[20] Jeffrey JP Tsai et al. A comparative study of formal verification techniques for software architecture. *Annals of Software Engineering*, 10(1-4):207–223, 2000.

[21] Justyna Zander et al, editor. *Model-Based Testing for Embedded Systems*. CRC Press, 2011.

[22] Lisa Brownsword et al. A case study in successful product line development. Technical report, Software Engineering Institute, Camegie Mellon University, Pittsburgh, Pennsylvania 15213, October 1996.

[23] Rick Convington et al. *Formal Methods Specification and Verification Guidebook for Software and Computer Systems*. National Aeronautics and Spcae Administration, Washington, DC 20546, July 1995.

[24] Roberto Cavada et al. *NUSMV 2.3 Tutorial*. Italy, July 2005.

[25] Sanjai Rayadurgam et al. Test-sequence generation from formal requirement models. In *High Assurance System Engineering*, pages 23–31. IEEE, 2001.

[26] Sanjai Rayadurgam et al. Auto-generating test sequences using model checkers: A case study. In *Formal Approaches to Software Testing*, pages 42–59. Springer, 2004.

[27] Steven P Miller et al. Formal verification of the aamp5 microprocessor: A case study in the industrial use of formal methods. In *Industrial-Strength Formal Specification Techniques*, pages 2–16, USA, 1995. IEEE.

[28] Mike Gordon. Background reading on hoare logic. Lecture Notes, April 2012.

[29] Hui-Min Huang. Autonomy levels for unammned systems (alfus) framework. Technical report, National Institute of Standards and Technology, October 2008.

[30] Farid Kendoul. Survey of advances in guidance,navigation and control of unmanned rotorcraft systems. *Journal of Field Robotics*, 29(2):315–378, 2012.

[31] NATO. Stanag 4586 (edition 3) - standard interfaces of uav control system (ucs) for nato uav interoperability, November 2012.

[32] Martin Ouimet. Formal software verification:model checking and theorem proving. Technical report, Massachusetts Institue of Technology, USA, 2008.

[33] Pnueli. The temporal logic of programsal. In *Foundations of Computer Science, 18th Annual Symposium*, USA, October 1977. IEEE.

[34] Klaus Pohl. *Requirements Engineering: Fundamentals, Principles and Techniques*. Springer-Verlag Berlin Heidelberg, Berlin Heidelberg, 1 edition, 2010.

[35] Chris Rupp, editor. *The Requirements Template - The Assembly Plan of a Requirement*. SOPHIST, 2010.

[36] Sadhukhan, editor. *Model Based Testing Practices. Idea, Approach and Solution*, Kolkata, India, 2011. Global Business Services (GBS).

[37] Daniel Kroening et al Vijay D'Silva. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, July 2008.