2001

Relational specification as a testing oracle

Harjinder Sandhu
West Virginia University

Relational Specification as a Testing Oracle

Harjinder Sandhu

Thesis submitted to College of Engineering and Mineral Resources
of
West Virginia University
in Partial Fulfillment of the Requirements
for the Degree of

Masters of Science
in
Computer Sciences

Department of Computer Sciences and Electrical Engineering

2001
Morgantown, West Virginia

Key words: Certification, Specification Based Testing, Relational
Specification, Testing Oracle

Relational Specification as Testing Oracle

By
Harjinder Sandhu

(Abstract)

Software engineering community is well aware of the usefulness of formal methods for specifying, designing and testing of the software. Despite this testing literature rarely deals with specification based testing. Testing from formal specifications offers a simple, structured and more rigorous approach to the functional tests than testing techniques. An important application of specification in testing is providing test oracles. The rise of use of computers in control safety critical systems, i.e., flight control systems, necessitates that rigorous system testing is performed before the deployment. In flight control systems, requirements are mostly concerned with the safety and maneuverability of an aircraft. In this domain, the use of formal approaches to requirements specification and system verification is strongly encouraged. In our study relational notation was used to model the requirements of generic flight control system. The advantage of relational approach is that the requirements can be partitioned into less complex components. Each component is separately specified with a set a relations. The formal aspect of the relational notation is exploited in a verification framework where the specifications are used as an oracle to test a system implementation.

To
My Brothers
and
Sister

Table of Contents

Acknowledgements

I adequately express my gratitude to my supervisor, **Dr. Bojan Cukic**, whose enthusiasm, encouragement, and wisdom have been invaluable to me. Moreover, I wish to thank him for letting me find my own way, being there when I needed help, and for always having the right words.

I am very thankful to **Dr. Ali Mili** for providing me the reference material when ever I needed. He always welcome me with the helping hand whenever I approached him. I am heart fully thankful to **Dr. Vittorio Cortellessa** for his tireless and thorough efforts during the course of this investigation and preparation of the manuscript.

Appreciation is also extended to my friends **Rajesh, Anil, Rekh, Sujay, Balaji, Gopal, Karthik, Sekhon,** and **Vinod**   for their help and pleasant companionship. I greatly appreciate the brotherly love provided by **Dr. Bhumbla, Amandeep, Ramkumar** and **Swarn** during this course of study.

I can never express enough appreciation to **my parents, brothers, sister, sister in law**, and my wife **Deepak** for their ever lasting love, encouragement and support during this course of study.

List of Figures

Chapter 1

Introduction

1.1 Motivation and Objectives

Software testing is the process of executing a software system to determine whether it matches its specification and executes in its intended environment. Software testing is arguably the least understood part of development process [33]. Every software development organization tests its products, yet delivered software always contains residual defects of varying severity. Sometimes it's hard to imagine how a tester missed a particularly glaring fault. Despite the major limitation of testing that it can only show the presence of errors and never their absences, it will always be a necessary verification technique.

The software engineering community is aware of the usefulness of formal methods for specifying and designing software. The accepted role of formal specifications in program verification is as the basis for proofs of correctness and rigorous transformation methodologies. However, formal specifications can play an important role in software testing. Of course, it is not surprising that specifications are important to software testing; it is impossible to test software without specification of some kind [38]. As Goodenough and Gerhart [8] noted that testing based only on program implementation is fundamentally flawed. Despite this, only a small portion of the testing literature deals with specification based testing issues. In testing, informal specification have limited usefulness (but are still required) but the real benefits are to be gained from formal specifications, which are now reaching a level of maturity and stability.

In this thesis we examine applications of formal methods to software testing which offer many advantages for testing. The formal specification of a software product can be used as a guide for designing functional tests for the product. The specification precisely defines the fundamental aspects of software, while more detailed and structural information is omitted. Thus, the tester has the important information about the product's functionality without having to extract it from implementational details. Testing from formal specifications offers a simple, structured, and more rigorous approach to the development of functional tests than standard testing techniques. An important application of specification in testing is providing test oracles. The specification is an authoritative description of system behavior and can be used to derive expected results of the program execution test data.

The rise in use of computers to control safety critical systems, i.e., in reactive systems and flight control systems, necessitates that rigorous software testing is performed before deployment . Traditional testing techniques in validation of concurrent systems fail as components interact with each other in several different ways. For these type of system alternative certification methodologies are needed based on the combination of mathematical proof and testing. Using formal methods to specify and verify high assurance systems is becoming a major thrust as system designs become more complex. Flight Control Systems (FCS) requirements are mostly concerned with the safety and maneuverability of an aircraft, and consequently, with the safety of crew and passengers. In this domain, the use of formal approaches to requirement specification and system verification is strongly encouraged. In our study relational notation was used to model the requirements of generic FCS. The advantages of relational approach are that the requirements are partitioned into less complex components. Each component is separately specified with a set of relations.   The formal aspects of the relational notation is exploited in a verification framework where the specifications are used as an oracle to test a system implementation. So the objectives of this study are

1) To demonstrate the issues that need to be considered when using relational notation for requirement specification

2) To demonstrate the treatment of quantifiers in the relational specification with reference to testing

3) To build a framework that exploits relational specification in the verification environment

1.2 Thesis Outline

The thesis is organized as follows. Chapter 2 is reviewing the literature related to the thesis, providing an understanding of specification based testing , benefits and limitation of using formal methods in testing, test oracles and the issue of software and system certification. Chapter 3 is the description of relational approach, how it is used in testing and description of the quantifiers that can be exploited in testing. Chapter 4 illustrates the application of the approach to the case study and *Adaptive Fault Tolerant Flight Control System***,** the tool that was used to create simulation test environment, and the discussion of the results of verification. Chapter 5 concludes the thesis and points some to future directions for research on the relation notation. Appendix A contains the description of input and output that was required by the simulator. Appendix B contains the plain English and relational requirements for flight control system.

Chapter 2

Related Work

Software Testing is defined as the process of exercising or evaluating system or a system component by manual or automated means to verify that it satisfies requirements or to identify difference between expected and actual results [IEEE89]. Software engineering community is well aware of the vital role that testing plays in software development. Testing is a practical means of detecting program failures that can be highly effective if performed rigorously. Despite the major limitation of testing that it can only show the presence of faults and never their absences, it will always be a necessary verification technique. Two basic approaches to software testing are *specification based testing* and *program based testing.* Research on testing indicates that both approaches are needful for fault detection [4, 29].

2.1 Program Based Testing

For the long time in the software industry, most software testing experts believed testing should be based on the code, profiles of the software under test, or on statistics about the software. The traditional development process is a process in which program objectives are transformed into program requirements, then into design specifications, implemented into code, tested and finally placed and maintained in operational status [4]. While this methodology gives the tester an idea of desired behavior, it does not tell the tester whether software works as specified. To judge if software behaves as expected, tester must know the requirements on which to base tests and results. Another problem with using this methodology at the end of the software lifecycle is that testing is often forsaken for delivery date commitments. The life span of a software product, however, realistically involves multiple versions of the software. Over a single production cycle, a product may have to be tested several times . In many cases, software development and problem fixing take the majority of the time in the schedule. Often, this means not enough time is devoted to testing [34]. A new testing methodology that resolves some of the problems started above is *specification based testing.*

2.2 Specification based Testing

Generally, there are two types of specification *operational or model based specification* describes the system's operating rules. Descriptive *or property based specification* describes the required system properties. A descriptive approach is especially important during the early phases of the software development when the objective is to describe precisely what the system must do rather than how system is to be implemented. Although most formal specifications cannot be classified by purely operational or descriptive, the difference between the two styles does not suggest their relative merits. Generally, operational specifications are closer to designer intuition, are easier to develop, can bias designer towards particular solutions. On the contrary, descriptive specifications usually have less implementation bias and they are more abstract.

Specification based testing is based on a program's requirements. When the requirements are specified, the design, coding and testing phases are all based on the requirement specification. Tests are created directly from the requirements during the designing and coding stages rather than after the coding stage. A comparison between program based testing and specification based testing is shown in the Figure 1.

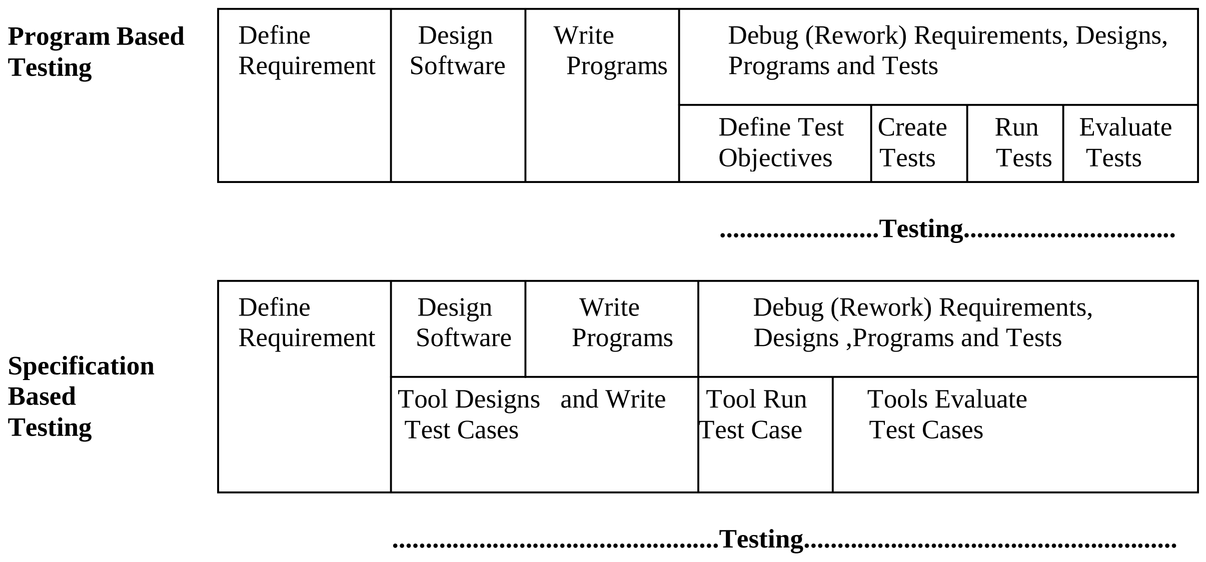

Program Based Testing
Define Requirement
Design Software
Write Programs
Debug (Rework) Requirements, Designs, Programs and Tests
Define Test Objectives
Create Tests
Run Tests
Evaluate Tests
........................Testing................................
Specification Based Testing
Define Requirement
Design Software
Write Programs
Debug (Rework) Requirements, Designs ,Programs and Tests
Tool Designs and Write Test Cases
Tool Run Test Case
Tools Evaluate Test Cases
....Testing...

Figure 1: Program vs. specification based testing

So, specifications can play an important role in software testing. One can argue that it is impossible to test software without specification of some kind. As Goodenough and Gerhart [8] note, testing based only on program implementation is fundamentally flawed. Despite this, only limited portion of literature deals with specification based testing issues. Testing on the basis of formal specification is beneficial when compared with testing based on the informal specification because informal specifications are ambiguous in some situations.

Formal specifications are of great importance in testing, for they determine what the software ought to do and must necessarily form the basis for the testing of the functionality of the software. The following has been observed by Richardson *et. al* [24]:

'Current software testing practices focus, almost exclusively, on the implementation, despite widely acknowledged benefits of testing based on software specification.' Considering information from formal specifications enables testing intended behavior and actual functionality. Specification based testing techniques may direct attention to aspects of the problem that have been implemented incorrectly or completely neglected, while implementation based techniques reveal such aspects by chance.

Real time systems are often safety critical. Ensuring the correctness of the these system is very important. Since real time systems are also typically complex , with behavior depending on the inter-relationships among the timing of the events of the system, testing is inadequate as the sole means of ensuring the correctness of such systems. *Formal methods* offer the hope of guaranteeing the correct behavior of such systems. *Formal methods* in general, refer to the development method based on some formalism. Thus a proof method for proving mathematical correctness cannot be formulated independently of the algebraic or logical formalism that supports it. Once a system and its implementation have been stated in a suitable notation, that the system satisfies the properties can be proved as a mathematical theorem. The consequence is that the level of confidence typical of mathematical reasoning can be obtained for engineering systems. In developing the systems, formal methods can have an impact, in principle, on any single phase (i.e. specification, implementation etc) as well as on the whole life cycle. For that different formal methods have been proposed and used, based on mathematical logic(ASTRAL[Coe 94]; RTL[Jah 86]), graphical notations, state chart like notations) [Jah 94, Ost 92] , petri -net based tools [Cuk 98] state machine models, timed automata [Lyn97] and verification based on model checking [Cla 86] and process algebra. The use of formal methods in the early phases of software life cycle, i.e., specifications are important because of the following reasons:

- The early phases in the system development are the most important because rest of the development cycle build on them.
- Despite their importance, the early phases are not well supported by high quality methods and tools. Moreover unlike later phases, one can rely on good programming languages and environments, in the early phases , much is left to the designers' intuition and common sense [6].

In software development formal specification can be used by

- *designer,* to formulate and experiment with the design of the system,
- *implementor,* as a precise description of the system being built, particularly if there is more than one implementation,
- *documentor,* as an unambiguous starting point for user manuals,
- *quality control* for the development of validation strategies.

Using a specification, the designer of a system can reason about properties of the system before development starts; and during the development, formal verification that an implementations meet its specification can be carried out. When an existing system is being specified there are both short and long terms benefits. In short term, defining the specification uncovers those parts of the system that are either incomplete or inconsistent. They give insights into anomalies in the existing system and can suggest ways in which the system could be improved.

In the long term the specification could be used

- for re-implementation of all or part of the system,

- as a basis for discussing and developing specifications for changes or additions of the system,
- to provide a model of the functional behavior of the system suitable for educating new staff.

Re-implementation may involve a new machine architecture, programming language or operating system or a restructuring to take advantage of multiprocessor or distributed system. As the specification is implementation independent, it provides a suitable starting point for each of the above alternatives when changes or addition to the system are to be made, new specification can be developed with reference to the previous specification. These developments will give insights into the effect of the changes and their interaction with existing part of the system. As the specification is a formal document, it provides a more precise description for communication between the designers that the natural descriptions. This should help to reduce the misunderstanding among the people involved.

Experiments with specification provide a quicker and cheaper method of investigating a number of alternative changes to the system than implementing the changes. On the other hand, because the specification is implementation independent, it cannot provide answer to questions of how difficult the changes will be to implement or their impact on the performance of the system. However, as it is at high level of abstraction it can give a better insight into the interaction of changes with other components of the system. It is just these high level interactions which get lost in the informal specification and in the detail of implementation. While working predominantly at a more abstract level the specifiers must be experienced in implementation and should be aware of the implementation consequences of their decisions. Those parts of specifications for which the implementation consequences are unclear should be further investigated before detailed implementation is begun.

2.2.1 Benefits and Limitations of Formal Methods

The experience of formal methods in industry and academia suggests that although formal methods can bring benefits to software development, many limitations exist. Austin and Parkin [2] summarize the benefits and limitations of formal methods in terms of safety critical system applications.

The perceived benefits of using formal methods are:

- Requirements and specifications are unambiguous. The main reason for this is two folds. The first is that all the variables used in formal specifications are typed and each type definition is based on the mathematical objects (e.g. natural number, real number, or boolean value) that have precise semantics. The second reason is that every operation in formal specification is defined precisely in the sense of its precise input and output relationship.
- Errors due to misunderstanding are reduced. As formal specifications are unambiguous, communication between people involved in requirement analysis, specification construction, design and implementation via the formal specification is enhanced. Implementations based on formal specification are usually easier that those based on the informal ones: This is because formal specifications usually present precise tasks for implementation, whereas informal ones cannot easily do so.

- Correctness proofs can be carried out, especially for safety critical properties. Correctness proofs have been recognized as a powerful approach to verifying implemented software against their specifications. They are especially important for safety critical applications. Since formal specifications adopt mathematical notation, correctness proofs become possible.
- Validation of requirement specifications becomes easier. Because of the precision of formal requirements specifications, every task specified can be precisely interpreted thus enhancing the clients' ability to scrutinize the correctness of formal requirements specifications.

The perceived limitations of the formal methods are:

- Formal specifications are difficult to read. The first reason for this is that the majority of people working in the computing industry at present are accustomed to traditional f methods and not well trained in formal notations. The second reason is that mathematical notations are usually more difficult to understand than informal descriptions. Two elements contribute to this difficulty. First, mathematical notations are concise and the information described by them is therefore compact. Second , the language in which these descriptions are expressed is necessarily terse and populated with abstractions.
- Formal methods cannot help model all aspects of real world. The difficulty is that the real world includes static and dynamic aspects while formal methods are a static technology for dealing with modeling and abstraction. Dynamic aspects may be

modeled using formal methods, but the model produced cannot really demonstrate the dynamic behavior of the system.

- Correctness proofs are resource- intensive. This is because considerable time is required to produce formal specifications. Furthermore, since there is intrinsic difficulty in performing correctness proofs automatically (e.g., assertion construction, associated knowledge management and efficient use), proofs have to be done manually or interactively with machines, which is resource intensive.
- Development cost increase. The main reason for this limitation is that many companies and projects need to invest more money for training their staff in formal technology.
- Formal specifications can still contain error. As mentioned earlier formal methods can help reduce errors due to misunderstanding. However, this is no guarantee that people will not make mistakes in formal specifications (e.g., syntactic errors and semantic inconsistency). No formal method so far can provide automatic support to semantic consistency checking for formal specifications (only some tools for syntax and type checking are available)
- There is no mechanism available in many popular formal methods for describing time constraints on a proposed system or a component of the system: The reason for this is that the initial purpose of some formal methods(e.g. VDM, Z) was not the development of time critical systems but the development of non time-critical systems. However, if they are used for time critical applications, it becomes difficult or impossible to describe timing behaviors.

- Environments to support the use of formal methods are not readily available. Tool to support the use of formal methods merely exist. None of the existing one is powerful enough to support the whole activity of using formal methods, such as consistency checking of specifications, specification refinements, correctness proofs, software testing etc.
- It is not yet clear how to incorporate formal methods into the whole life cycle of software development. To solve this problem, there is a need to answer the following questions: How to construct good quality formal specifications (understandable, consistent, and structured)? How to refine formal specifications? How to perform software verification (including correctness proofs and software testing) in efficient way? How to write documentation? How to manage a software project ?

To extract maximum out of the use of formal method they should not only provide rigor and lead to increased system reliability. They must also provide some additional functionality including:

- *Readability***:** Generally formal documents have been criticized as hard to understand. To accommodate that problem formal methods should be supplemented with more intuitive notations such as annotated with explanatory comments. By doing this a formal document can be understood by those with little technical background.
- *Scalability***:** The method should scale up from small examples to large, complex real world systems and should be maintainable and reusable.
- *Tool Support***:** It is always better if the method can be supported by well-engineered tools. Many practically situations are unmanageable without some automation. The

basic principle behind this scenario is that user should take care of intuitive part of the task, while clerical data should be managed by the tool.

The formal specification of a software product can be used as guide for designing functional tests for the product. The specification precisely defines fundamental aspects of the software, while more detailed and structural information is omitted. So in zest we can say that testing from formal specifications offers a simpler, structured, and more rigorous approach to the development of functional tests than standard testing techniques. The strong relationship between specification and tests facilitates error pin-pointing and can simplify regression testing. Other benefits of specification-based testing include using the derived tests to validate the original specification, simplification of auditing of the testing process, and developing tests concurrently with design and implementation. An important application of specifications in testing is providing test oracles. The specification is an authoritative description of system behavior and can be used to derive expected results for test data .

2.3 Testing Oracle

A Test oracle determines whether a system behaves correctly for test execution. Test oracle provides a means for determining whether an implementation functions according to its specification. When a formal specification exists, it is logical to use that specification as a test oracle. In practice, however, the differences between specification languages and programming languages prevent the specification from being used directly as an oracle. Thus it is necessary to translate the specification into a form that can be readily used as testing oracle.

DAISTS (Data Abstraction, Implementation, Specification and Testing System) is a system which focuses on using specifications as oracles [36]. The DAISTS approach is to annotate program code with algebraic specifications of data types and tests. The specification axioms are translated into code segments which call procedures in the implementation. The tests specify which axiom they are testing and provide instantiations for the free variables in the axiom. DAISTS checks that the program implements the specification for the cases defined in the tests section by constructing implementation driver from the specification and using the tests as input. This notation of the specification driving the implementation was extended by Hayes [35], who considered oracle issues for model-based specifications of abstract data types. Hayes [35] also showed how the oracle procedures can be derived from Z specifications of abstract data types to check invariants, pre-conditions and the input-output relationships.

Murray et al [39] identify two types of test oracle. An *active oracle* implements the expected behavior of the software under test. A *passive oracle* checks the behavior of the software, but does not produce it. Peters and Parnas [21] developed test oracle from the design documentation. Their oracles were precise and relatively readable, written in terms of data structure using a relatively expressive notation. Their generated oracle meets the following criteria:

- For any test results (input, output pair) the oracle can be used to determine whether or not the program under test satisfied the specification.

- It can be used to determine whether or not the specification allows termination for a particular test case, and if it does, if the program is required to terminate for that case.
- It does not require pre calculated "expected results" for the test cases.
- It does not require that there be a unique correct answer.
- It does not assume the existence of a previous version of the program under test that can be assumed to be correct..

Research about temporal logic specifications has focused on model checking [3, 31] which determines whether it is possible to violate the specification on any execution, but requires exploration of entire state space . Specification based oracles, on the other hand, only determine whether a particular execution violates the specification, but do not need to explore the entire state space . Specification based oracles can be thought of as consisting of a general purpose oracle procedure for a type of oracle, and information specifying desired behavior for specific test case or a specific component under test [25]. Malley et al [22] reported that general approach for specification based oracles is to convert a formal specification of required behavior into an internal representation (the oracle information) and then develop the checker for that internal representation(the oracle procedure). Richardson et al [25] developed an approach to deriving testing oracles from specification for reactive systems and incorporated these oracles in the testing process. This approach is workable for a wide variety of specification formalisms and, hence, is useable for a variety of application domains, behavioral aspects, and computational paradigms. They derive test oracles from multi specification and compose these multiple oracles to check test results for behavioral correctness. In addition they couple oracle derivation with testing and represent an oracle for each test class specified by the testing criteria in use. They also examine the associated monitoring needs. To enable comparison to test results to specification based oracles, their approach hinges on a mapping between the abstract specification and the concrete implementation.

2.4 Intelligent Flight Control Project

In this section we describe the case study considered in the thesis. In life-critical and mission-critical applications, it is important to make provisions for fault tolerance to take into account a wide range of contingencies. Whereas fault tolerance is usually equated with duplication-based redundancy, in this project, the team chose to adopt an approach based on analytical redundancy. Analytical redundancy relies on relations that hold naturally between system variables, and attempt to exploit them for the purposes of fault tolerance. The following observations support our analytical redundancy approach:

- U.S. Air Force accident reports regularly report cases where an aircraft has crashed after losing control surfaces, despite having the physical means to fly on and land safely [17,18,19].
- In a non-fatal accident that occurred at DFW in March 1997, the aircraft (a Boeing 767) lost 18 feet of the right outboard flap, but the pilot could regain control and land it safely by using the left aileron [20].

- It is widely believed, from accident reports, that the fatal accident of Alaska Airlines Flight 261 (an MD-80) on January 31, 2000, could have been averted despite the loss of stabilizer controls, had the pilot used the other controls appropriately.

All these instances show that it is possible for a flight to survive the loss of some flight surfaces, or the loss of control over some flight surfaces. The key, it seems, is to recognize that these losses produce a different control law, and to operate the aircraft according to the new law. Of course, the range of possible control laws that result from the losses is infinite, and ranges over a continuum -hence it is unrealistic to expect pilots to be exhaustively trained on these. In this project, a Flight Control System (FCS) that is fault tolerant with respect to sensor faults has been considered.

2.4.1 The Issue of Certification

A major issue in the use of these adaptive fault-tolerant FCSs is certification. Current practices rely heavily on testing. For example, the Federal Aviation Administration (FAA) standard for avionics software uses the modified condition decision coverage (MC/DC) criteria, which is based on demonstrating, for each branch in the software, that each parameter can independently affect the result. While this is less demanding than full branch condition coverage, it presents a huge cost overhead for large avionics systems. Boeing estimates that 40% of the software development costs for the 777 were spent on testing.[1] Hence, testing to this standard has become a major cost driver in the development of new aircraft. One consequence is that alternatives for flight

[1] The total development cost for the B777 was \$5 billion. Approximately half of this was software development with roughly a billion dollars spent on software testing.

software certification could enable a significant reduction in the cost of flight control software development.

At the same time, this form of testing has a number of important limitations. First, it is accepted in software engineering that “for complex systems, testing can never demonstrate correctness; it can only be used to reveal errors.”[2] Second, criteria such as MC/DC are based on the structure of the code, and hence may not uncover problems associated with missing or incorrect requirements; neither can they uncover systemic problems to do with the interaction between components. Finally, these criteria are meaningless for new software technologies, including adaptive controllers . For these types of systems, alternative certification methodologies are needed, based on a combination of mathematical proof and testing to system and performance requirements.

2.4.2 Avionics Software Certification Issues

As mentioned in the section 2.4.1 certification is the major issue in avionics software. In this section we will discuss the previous work done on the certification standards. *Certification* can be defined as an official assessment of equivalence between the specified and the actual service provided by the software and/or system [27]. There are basically two ways of certifying an embedded software-based system: the indirect “process certification” and the promising “product certification”. Process certification is an indirect way of certification. It works as follows:

- The certifier and the developer agree on a development methodology, the stringency of which is a function of the criticality of software,

[2] E. W. Dijkstra, quoted from “Art of Software Testing,” by Glenford J. Myers, (February 1979), Jon Wiley & Sons; ISBN: 0471043281

- This methodology is applied by the developer,
- The certifier checks that the methodology is effectively applied.

Process certification is the core of current standards for software certification. The Federal Aviation Administration (FAA) and other federal agencies, such as the Nuclear Regulatory Commission (NRC) and Food and Drugs Administration (FDA), have chosen to perform software certification using a technique similar to that used for certifying hardware. The basic message of the Radio Technical Commission for Aeronautics document RTCA/DO-178A [23], for example, *"is that designers must take a disciplined approach to software: requirements definition, design, development, testing, configuration management, and documentation."* There are two observations leading to conclusion that process certification is insufficiently rigorous for intelligent flight control systems:

1. *General observation*: Software engineering techniques for building and validating software for complex embedded systems so that it adheres to stringent safety and reliability requirements are the subject of permanent improvements and represent open research problems.
2. *Specific observation*: Intelligent flight control systems or, more generally, systems built by any soft computing paradigm, are adaptive; i.e., they change over time. While this is one of the basic reasons behind the technical appeal of soft computing platforms, it implies that process certification procedures are inadequate. In other words, the adequacy of the process does not imply the adequacy of the system, since the system will change over time.

For flight control systems, certification is a mandatory process that must be completed before commissioning the product. The benefits are evident. Product certification is defined as the direct assessment of the adequacy of the actual service versus the specified service. Software product certification, shown in Figure 2, comprises of three steps:

1. Precise *identification*, at the specification level, of the functional characteristics of software and of the non- functional attributes necessary for its intended use: reliability, maintainability, security, etc.
2. *Measurement* (quantitative) and *examination* (qualitative) of the same functional attributes at the specification, software (product) and service level.
3. Multi-dimensional *assessment* of the equivalence between the two sets of attributes.

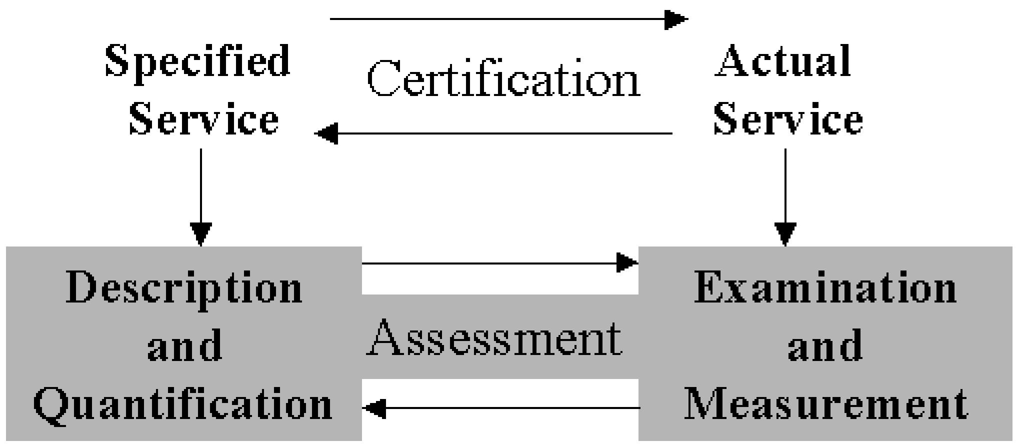


Figure 2: Software Product Certification

For neural network based flight control schemes, theoretical foundations for product certification have yet to be established. The basic principles governing the software life cycle processes for aircraft control systems, laid down in the RTCA/DO-178B document, were designed with the traditional (algorithm) computing paradigm in mind. In spite of traditional computing paradigm there are some strengths that are associated with RTCA/DO-178B. Also, the adaptive ("non programmed") flight control software requires rethinking or modification of certification procedures proposed in this certification standard. The strengths of DO-178B can best be summarized as follows:

- Development process for the DO-178B is well defined and orderly explained.
- Constraints and standards (for requirement, design, and coding) can impact verification effort.
- Organization and tracking tools mention in the DO-178 B can significantly increase efficiency.
- Automation can significantly speed up paper dependent processes.
- As the process of certification by DO-178B is systematic, generally high quality product can be produced.

Limitations of existing certification standards are:

DO-178B has been scrutinized as a standard, and a number of weaknesses have been reported. Some of the well-known limitations include:

- Lack of examples/suggestions creates difficulty for first time developers (but they are not intended audience anyway).
- Compliance with guidelines requires an extensive effort.

- No standardized tools have emerged, and some of the existing tools can be harmful.
- Time and money to learn new tools are substantial and should be planned for.
- People involved with the output from the tools must understand the tools, which is not always simple to achieve.
- Evidence supporting that an objective is achieved not clearly identified.

FAA identified further limitations of DO-178B:

- Inadequate and ambiguous guidance for requirement definition and analysis.
- Inadequate guidance for partitioning.
- Inadequate guidance for verification activities (adequacy of coverage criteria, suitable analysis techniques…).
- Inadequate guidance on COTS software.
- Inadequately addresses the effect of software to the safety of the overall system.

Chapter 3

Relational Approach to Software Specification

3.1 Relational Background

Relation is a set of tuples. If the set consists entirely of pairs (2-tuples), the relation is known as binary relation. The set of values that appears as the first element of pair in a binary relation is called the domain of the relation. The set of values appearing as the second element of a pair is called the range of a relation. A binary relation on set *S* is a subset of *S X S*. Let (*s*,*s'*) be an element of relation R . Then *s* is said to be an antecedent in *R* , while *s'* is said to be an image in *R*. The set of images of *s* in relation *R* is denoted by *s• R*, and set of antecedents of element *s* by relation *R* is denoted by *R• s*. The set of all the antecedents of relation *R* is called the domain of relation *R* and denoted by

$$dom(R)= \{ s \mid \exists s':(s,s')\in R)\}$$

The set of all images of relation R is called the range of relation R, and denoted by

$$rng(R)= \{ s \mid \exists t:(t,s')\in R)\}$$

Among the relations on a set following relations are important:

- *Universal Relation:* defined as *S X S* and denoted by L(S).
- *Identity Relation:* defined as $\{ (s,s') \mid s\in S \wedge s'=s\}$ and denoted by I(S).
- *Diversity Relation*: defined as L(S)/I(S) and denoted by V(S).
- *empty relation:* defined as the empty set and denoted by ϕ(s).

As a relation is a set, we can perform on relations all the set theoretical operations, such as complement, intersection, and union. In addition, we define the product of two relations *R* and *R'* as

$$R \circ R' = \{(s, s') \mid \exists t : (s, t) \in R \wedge (t, s') \in R'\}.$$

Constant relations on *S* include the empty relation, and the full relation. Heterogeneous relations, from a set *A* to set *B*, can be defined similarly, and could in fact be considered as relations on *S*, for

$S = A \cup B$.

Predicate is a template that describes a property of objects or relationship among objects represented by the variables. More rigorously, a predicate is a relation. Large and complex sentences are constructed in predicates by using the connectives. Though there are many connectives the five basic connectives are NOT, AND, OR, IF-THEN (or IMPLY), IF-AND-ONLY_IF. They are also denoted by the symbols :$\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$ respectively. A predicate with variable can make a proposition by applying one of the following two options to each of the variables:

- assign a value to the variable,
- quantify the variable using a quantifier.

In general, quantification is performed on formulas of predicate logic, such as *p(x)* by using quantifiers on variables. There are two types of quantifiers: *universal quantifier* and *existential quantifier*.

3.1.1 Universal Quantifier

The expression $\forall x\ p(x)$ denotes the universal quantification of the *p(x).* Translated into English language, the expression is understood as *"For all x, p(x) "* or *"For every x, p(x)"*. $\forall$ is called the universal quantifier and $\forall x$ means all the objects *x* in the universe. If this is followed by *p(x)* then the meaning is that *p(x)* is true for objects *x* in the universe (set of objects of interest) i.e. domain of the individual variables. If all the elements in the universe of discourse can be listed then the universal quantification $\forall x\ p(x)$ is equal to the conjunction : $p(x_1) \wedge p(x_2)........\wedge p(x_n)$.

3.1.2 Existential Quantifier

The expression $\exists x\ p(x)$, denotes the existential quantification of *p(x)*. Translated into the English language, the expression could also be understood as : *"There exists an x such that p(x) "* or *"There is at least one x such that p(x) "*. $\exists$ is called the existential quantifier and $\exists x$ means at least one object *x* in the universe. If this is followed by *p(x)* then the meaning is that *p(x)* is true for at least one object of the universe. If all the elements in the domain of interest can be listed then the existential quantification $\exists x\ p(x)$ is equivalent to the disjunction: $p(x_1) \vee p(x_2)........ \vee p(x_n)$.

A variable in a predicate is said to be *bound* if either a specific value is assigned to it or it is quantified. If a variable is not bound, it is called a *free* variable. The extent of the application of a quantifier is called the *scope* of the quantifier and is indicated by square brackets [ ]. If there are no square brackets, then the scope is understood to be smallest well formed formula following the quantification.

For example, in $\exists x\ p(x,y)$ the variable *x* is bound while *y* is free.

In $\forall x\ [\ \exists y\ p(x,y) \vee Q(x,y)]$, *x* and the *y* in *p(x,y)* are bound while *y* in *Q(x,y)* is free, because the scope of $\exists y$ is *p(x,y)*. The scope of $\forall x$ is $[\ \exists y\ p(x,y) \vee Q(x,y)]$.

3.1.3 Order of Application of Quantifiers

When more than one variable is quantified, such as in $\exists y \forall x\ p(x,y)$, quantifiers are applied from the inside, that is one closest to the atomic formula is applied first. Thus $\exists y \forall x\ p(x,y)$ reads $\exists y[\forall x\ p(x,y)]$ and we say for some *y, p(x,y)* holds for every *x*. The positions of the same type of quantifiers can be switched without affecting the truth value as long as there are no quantifiers of the other type between the ones to be interchanged. For example $\exists x\ \exists y\ \exists z\ p(x,y,z)$ is equivalent to $\exists y\ \exists x\ \exists z\ p(x,y,z)$ *etc* . It is the same for the universal quantifiers. However, the positions of different types of quantifiers can not be switched for example $\exists x \forall y\ p(x,y)$ is not equivalent to $\exists y \forall x\ p(x,y)$. Let p(x,y) represent $x<y$ for the set of natural numbers as the universe, then $\exists x \forall y\ p(x,y)$ reads *"for every number x there is a number y that is greater than x"* which is true , while $\exists y \forall x\ p(x,y)$ reads "*there is a number y that is greater than any number"*, which is not true.

3.2 Relational Software Specification

At the system level, requirements specifications represents an abstraction of the system behavior. They contain all the required features of system, without involving any details about their implementation. The fundamental role played by the correct specification of the system requirements in the design process has been widely addressed in literature :

" ......No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later...."[38] "..... requirement inadequacies play a major and expensive role in the project failure" [7].

Requirements specifications are also a means of communication between customer, designer and verifier. In order to be a good means of communication they should be unambiguous. Furthermore, they should be maintainable, since they are often refined over time. Use of plain English requirements does not allow to have unambiguous maintainable requirements. This reinforces the need of formal specification language with a well defined syntax and semantic. The formal approach we will be using is based on the predicate logic[1] and relational algebra [14] .

Relational Algebra provides the framework for formal approach to requirements specification. Relations are set, besides union and intersection operators there are other operators that are more specific to relations. Among them, *join* and *meet* are important. *Join* represents the *sum of requirement information* and *meet* represents the *common requirement information.* Furthermore it is possible to define an ordering among relations to measure the relative strength of requirements. This ordering is called *refinement*. A relation is said to refine another relation if it has a larger domain and has a smaller image set on a common domain. The refinement relation is abbreviated by $R \sqsupseteq R'$ and the join is denoted by $R \sqcup R'$.

3.2.1 Specifying with Relations and their Exploitation in Testing

Generally, programs read input data from some input file or device, process it and return the results onto some output file or device. The *input space* of a program is the set of inputs that the program may read in. The *output space* of a program is the set of outputs. The *internal space* is the set of values that the program's variables may take during the execution. Th input and input operations of program perform mapping from the input space to the program's internal space and from the program's internal space to the output spaces [12]. Most importantly one needs to take into consideration the state transformation taking place in the execution of the program, i.e., relation between initial states (presumably the states after the input is read) and final state (presumably the state before the output is written). Specification can be defined if provided with

- A space , say *S*, typically defined by set declarations.
- A relation on space *S* , say R, typically defined as set of pairs (s,s') such that some property holds between s and s'.

Mili et al [13] reported that specification represented by the pair *(S,R)* prescribes requirements on a program whose space is *S* (acting space). The pair *(s,s')* is in *R* if (completeness oblige) and only if (minimality oblige) the user considers that *s* is a possible initial state and *s'* is correct final state for initial state *s*. So an implementation *P* is said to be *correct* with respect to specification *R* on *S* if and only if *P* is defined for all inputs in the domain of *R*, and for each such an input, *P* returns a value *s'* such that $(s, s') \in R$. For any initial state *s*, there may exist more than one final state s' in $s \bullet R$; *R* is arbitrarily non-deterministic and if the space *S* of a specification *(S,R)* is implicit from the context, the specification may be represented by the relation *R* alone. So, modeling the specification by relational algebra provides the following benefits:

- unambiguous means of communication between domain experts and specifiers
- pre-check of the requirements within the prose to relations translation process
- capturing of requirements in a traceable manner.

- composition of the requirements.
- checking for completeness and minimality of the requirements.

Below is the example showing the conversion of prose requirement into relational requirement. Prose requirement is

" The incremental slideslip angle shall not exceed 2 degrees from the trimmed values and lateral acceleration shall not exceed 0.03 g while at steady bank angle up to maneuver bank angle limit reached during normal maneuver with the AFCS engaged"

The relational requirement for this statement is

MSBT=

turb( ): time_type $\rightarrow$ Boolean;

ϕ( ): time_type $\rightarrow$ angle_type;

β_{trim}: angle_type

Constant terms

Acc_{ϕ} =1:[degree] angle_type;

A_{y_max}= 0.02: [g] accerleration_type;

$\Delta\beta_{max}$ =1:[degree] angle_type;

Quantified terms

t_1,t_2,t_3 :time_type;

Auxiliary terms

none

RSLF= { (m,c) | $\forall t_1,t_2 : t_1<t_2$(

$\forall t : t_1 \leq t \leq t_2$(

$$\text{turb}(t) = \text{OFF} \wedge |\phi t| \leq Acc_{\phi}) \Rightarrow$$

$$|\beta(t)-\beta_{trim}| \leq \Delta\beta_{max} \wedge |A_y(t)| \leq A_{y\_max})\}$$

Each formal requirement is supplemented with a header that introduces monitored, controlled, constant, auxiliary and quantified variables/values used in the specification. The domain (MRAH) and range (RAH) are specified in terms of monitored and controlled variables, respectively. Note in this case , there are only constant and quantified terms. Constant terms are constant quantities used within specification; their value is specified along with their declaration. Quantified terms are quantified variables needed within the specification and listed in the header in order to specify their type. Auxiliary terms are used to specify the notation, none in this case.

3.2.2 Relational Specification for Testing and Treatment of Quantifiers in Testing

Requirements represent ideal (correct) functionality of the system. But implementation of the requirements could results in different functionality of the system. So the concern over here is to establish whether implementation meets the requirements. This is achieved by testing in which the correctness of an output of the system is checked by the oracle built from relational specifications. Figure 3 demonstrates how relational specification are used in verification frame.

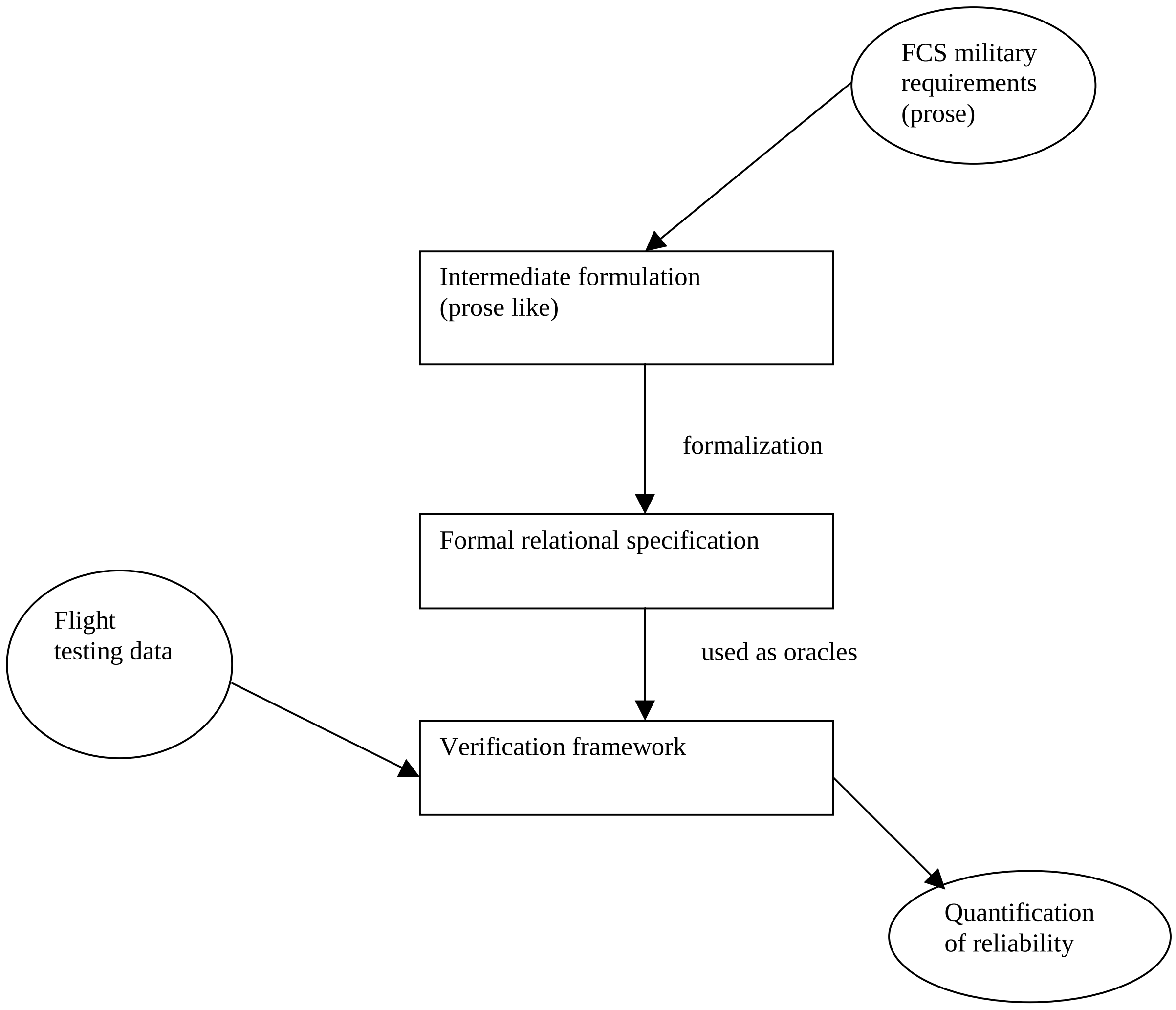

FCS military requirements (prose)
Intermediate formulation (prose like)
formalization
Formal relational specification
Flight testing data
used as oracles
Verification framework
Quantification of reliability

Figure 3: Use of Relational Specification for Testing

It is always better to defined the prose requirements in the intermediate notation. The goal of defining the requirements in an intermediate notation is to gain precision and understanding. Additional level of formalism allows more accurate analysis of the specification and highlights any existing ambiguity and/ or error. Then intermediate requirements are translated into formal relational representation. Then finally use formal specification are used as oracles, in the testing framework to check whether the implementation of the system fulfill the requirements. Circles in the figure represent process inputs and outputs while rectangles represent data processing.

It has been discussed above that quantifiers help to convert prose specifications into relational specifications which, in turn, is used as testing oracle to check the correctness of the application. But there are issues that need to be considered before using relational specification as testing oracle. Quantifiers, i.e., $\forall$ and $\exists$, used in relational specifications deals with time .Generally, time is not necessarily a main issue when dealing with relational specification but in the our system specification time is an inherent component because in our case study we are dealing with intelligent flight control system. In intelligent control system at each time interval the control system takes the snapshot of the sensor readings, process them and compute the actuator values and then waits for the next time interval to do the same. As in this case study we are studying a flight control system, which is a real time application, the issue of $\forall$ and $\exists$ quantifiers with respect to time is quite important. So in following discussion we are emphasizing the treatment of time with respect to the quantifiers.

3.3 Quantifiers with respect to time

In the specifications (Appendix B) time is considered continuous. But to make it practical in the sense of testing we have to consider it as a discrete variable. Considering continuous time results in trajectories that require only single input record and single output record as the result of processing. The data structure produced after considering continuous time shown in the Figure 4.

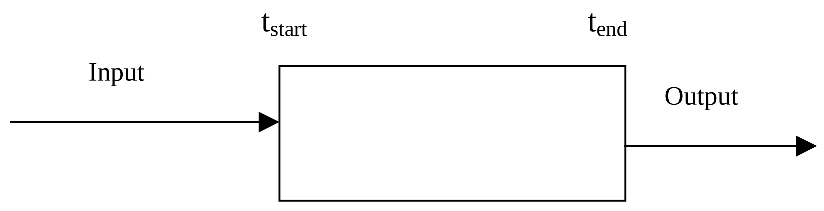


Figure 4: Data Structure After Considering Continuous Time

where t_{start} is the starting time and t_{end} is the end time

To use the relational specification as a testing oracle, a record like this used as an input to the fault tolerant capability requirement does not make sense. It is not feasible to meaningfully evaluate program correctness on a single record representing a potentially long trajectory. In this scenario testing quantified variables does not have well defined meaning due to single slot of time and single record of input and output variable. By considering this issue, thinking of time as a discrete variable makes more sense. In this case there is one input and one output record for each time interval provided by the program and it is appropriate to test all these records according to the preview criteria. To consider time as a discrete variable, flight simulator can be run off line and result in timed input/output file for each trajectory. The internal data structure is as shown in the Figure 5.

Input record 0 Output record 0	Input Record 1 Output record 1	…	Input record i Output record i	Input record i+1 Output record i+1	…	Input record n Output record n

Figure 5: Input/Output Workload to the Relational Specification

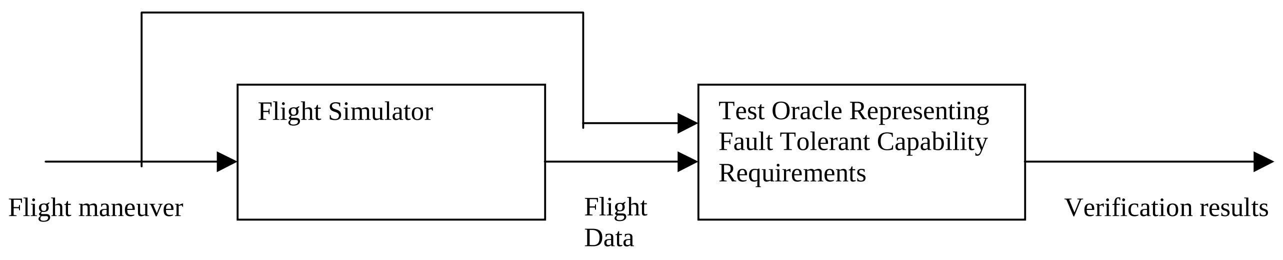

Flight Simulator
Test Oracle Representing Fault Tolerant Capability Requirements
Flight maneuver
Flight Data
Verification results

Figure 6: A Framework for FCS Verification

Value of inputs (i.e. parameters representing sensor readings) and output (i.e. control commands) variables for every time interval *dt* are stored in a slot of data structure. The data structure shown in the Figure 5 is input to test oracle representing fault tolerant capability tolerant block in the figure 6. This blocks checks whether the value of output produced by the simulator, providing input parameters representing sensor readings to the simulator, is within the domain delimited by those requirements and produces a boolean result according to that . List of all the relational specification is provided in the Appendix. Most of these specifications show the involvement of both quantifiers universal and existential, generally, in all of the relations universal quantifiers refer to time quantities only. The general format $\forall t : q(t)$ where q(t) is a condition.

In terms of file structure, these quantifiers refer to distinct (input/output) slots in the test data file. Upon assuming this, in quite a general case a relation assume the form

$\{ (m,c) \mid \forall t_1,t_2......t_k\ q(t_1,t_2......t_k) \Rightarrow p \}$

where q is a condition on time and p is a predicate on state variables.

Each of such relations is used as an oracle to test whether the system satisfies the requirement. This is done by analyzing $q(t_1,t_2......t_k)$ on the test file by checking in which slots a particular condition is true and then marking the slots where predicate is true by retrieving the appropriate entries from the file. For the universal quantifier the if the test succeeds for all values (and fail for none) then it can be deduced that (m,c) satisfies the oracle.

The treatment of existential with respect to time are handled by translating them to universal quantifiers by the relation.

$\exists t : p(t)$ is equivalent to $\neg \forall t : (\neg p(t))$.

The issues that need further attention is the number of input/output in the test data file (i.e. basically the size of the population) and the time interval between two data points.

3.3.1 Issue of Time Intervals

The choice of the value of time interval (*dt*) (shown in the above Figure 4) determines the time scale. It has to be handle differently and carefully with respect to both existential and universal quantifiers. In every time slot in the Figure 4 , the concern is the true and false value of predicate with respect to both existential and universal quantifier. So there are four cases that need to be considered

3.3.1.1 true value of predicate in at least one slot with respect to existential quantifier.

3.3.1.2 false value of predicate in every slot with respect to existential quantifier.

3.3.1.3 false value of predicate in every slot with respect to universal quantifier.

3.3.1.4 true value of predicate in every slot with respect to universal quantifier.

let's consider these cases one by one

3.3.1.1 true value of predicate in at least one slot with respect to existential quantifier :

When dealing with existential quantifier the concern is to find at least one value from the entire domain that satisfies certain condition or criteria. If there is chance of finding at least one value in the entire domain, the predicate is true against the testing oracle so the issue of time interval in no more an issue in testing.

3.3.1.2 false value of predicate in every slot with respect to existential quantifiers : Consider the scenario in the Figure 7 below where in every time slot the value of predicate against testing oracle is false.

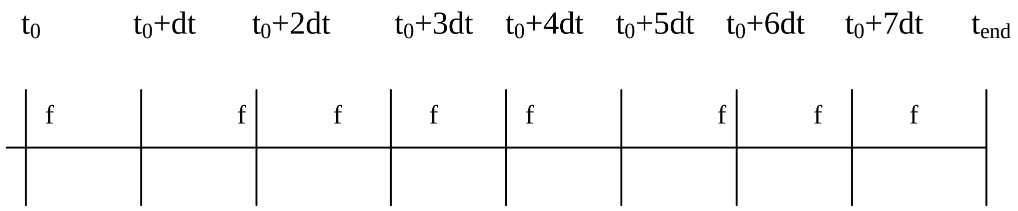


Figure 7 : False Value of the Predicate in Every Slot of the Input/Output File With Respect to Existenial Quantifier.

where, f represents the false value in every slot.

So for the existential quantifier the test fails for all the values, it can be deduced that the relation does not satisfy the oracle which may or may not be the case in real sense. The reason for that is explained in Figure 8 .

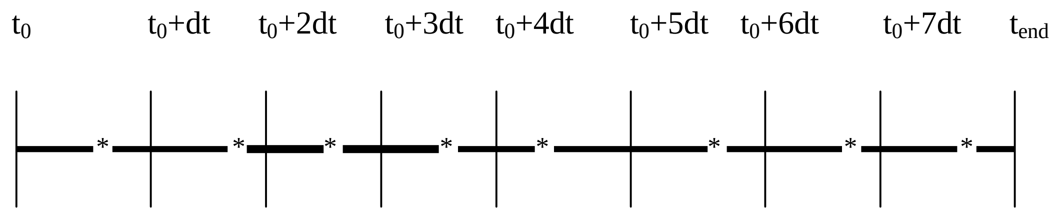


Figure 8 : False Value Pick From Each Slot

In the figure above * represents the points at which the value picked to test the oracle from each time slot and horizontal dark line (▬▬) represents the time interval from where no value is picked. As we are dealing with existential quantifier, concern is to find a single true value, to consider the relation satisfy the oracle, so the probability of true value falling between the two * points, i.e., on the horizontal solid line is more if the length of the time interval is more as compared to considering a small interval of time for that particular time.

3.3.1.3 False value of predicate in at least one time slot with respect to universal quantifier : The issue of time interval is not a concern when there is possibility of finding at least one time slot where the predicate is false. One can immediately deduced that relation does not satisfy the oracle.

3.3.1.4 True value of predicate in all the time slot with respect to universal quantifier: As the value of predicate in each slot is true one can think that predicate satisfies the oracle which may or may not true in reality. In this case we are also looking for at least a slot where the value of predicate should be false. So if the time interval between the two slots is more there is more chance of falling that false value into the interval which is not considered for testing.

So the issue basically is when the value of predicate is false in every slot with respect to existential quantifier and true in every slot with respect to the universal quantifier. So in these situation have to check the nature of variables, as these variables are domain specific need the expertise of the domain expert for making a decision regarding the time interval for those quantifiers. After getting information from the domain expert regarding the nature of the variable in question we can use the Shannon theorem to come up the value of the time interval for particular variable.

Chapter 4

Case Study and Results

A certification procedure includes precise *identification*, at the specification level, of the functional characteristics of software and of the non-functional attributes necessary for its intended use: reliability, maintainability, security, etc. It must proceed with *measurement* (quantitative assessment) and *examination* (qualitative assessment) of the same functional attributes at the specification, software (product) and service level. Therefore, a multi-dimensional *assessment* of the equivalence between the two sets of attributes must be a part of any certification environment. The assessment process proceeds by comparing specified service with actual value for qualitative factors. Figure 9 shows the general outline of the certification environment.

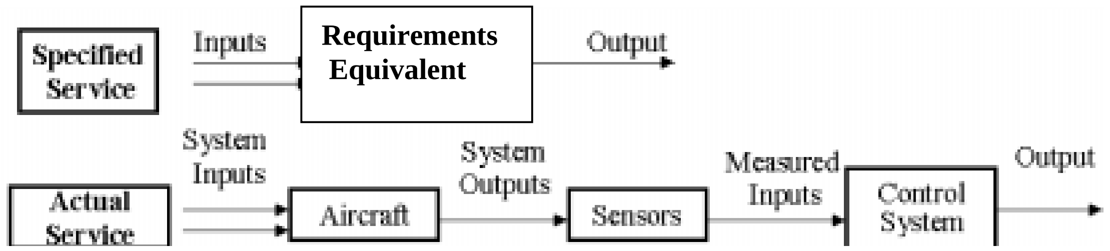


Figure 9: An Outline of the Certification Environment

4.1 Quantitative Assessment

Measurement is inherently the result of quantitative analysis. Quantitative analysis is the determination of the values of the quality factor of an object. This can be achieved by assigning a number to each quality factor. This assignment has to be done so that relations between objects are mapped homomorphically onto real numbers.

For all this, a *quality model* is required as basis. This quality model defines and decomposes the relevant quality factor and determines their interrelations. For description and measurement of specified service formal and semi-formal methods have been used. Structural analysis methods provide an intermediate level of formalism in the specification. Structured analysis supports the completeness and coherence verification but it doesn’t support proofs of correctness. For purposes of measurement of the specified services of this system executed equivalent of the specification was used.

4.2 Qualitative Assessment

Examination is the qualitative analysis and is done by analyzing the description (textual analysis), interpretation and execution. Textual analysis produces information on the syntax structure of the object and is conducted through *inspection* and tools of *static analysis*. Interpretation produces information on the algorithm that the object includes. Applicable methods are *inspectio*n, *symbolic execution* and *verification.* Execution requires executable objects and produces information on the dynamic behavior of the object i.e. on result, operation etc. Methods to be used are those of *dynamic analysi*s.

As indicated in Figure 9, certification environment includes measurement and comparison of the outputs of the specified service, represented by the any executable version of the formalized requirements specification, and the outputs of the actual service. In this study, the actual service was derived from the high fidelity simulation of the Beaver aircraft, enhanced to include the specifics of Sensor Failure Detection Identification Accommodation system, as shown in Figure 10.

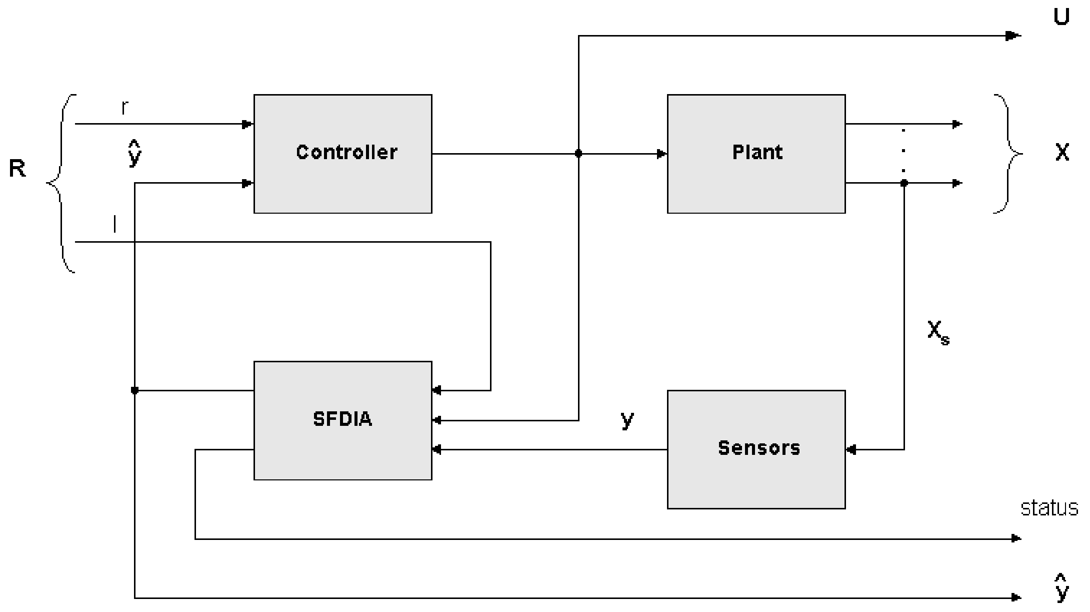


Figure 10: Inclusion of SFDIA into the Simulation of the Beaver Aircraft

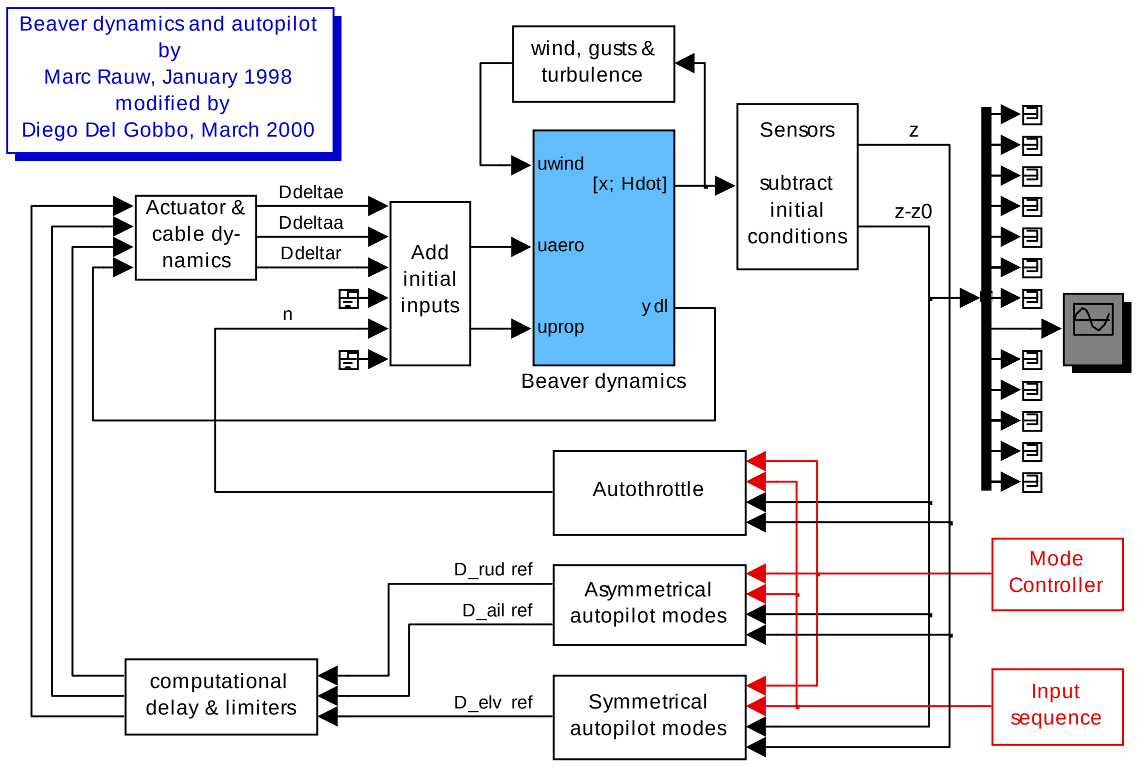
4.3 Tools for Testing

The Flight Dynamics and Controls (FDC) Toolbox, Version 1.2, was used to study the test case selection and test result interpretation for the system certification. This package provides graphical software environment for the design and analysis of aircraft dynamics and control systems, based upon MATLAB® and SIMULINK®. Test case generation and result interpretation were achieved by utilizing the Beaver Airplane simulator. The block diagram of the simulator is shown in Figure 11.

The top level of autopilot simulation model contains the following blocks:

- Beaver dynamics links the non-linear aircraft model Beaver to the autopilot simulation by means of an Simulink input and output block

- Symmetrical autopilot modes and Asymmetrical autopilot modes contain the symmetrical and asymmetrical control laws, respectively.
- Actuator and cable dynamics contain linear state space models of the dynamics of the actuators and the cables from the actuators to the control surfaces as used in the 'Beaver test aircraft.
- Computational delay and limiters takes into account the computational delay in the evaluation of the control laws and the input limitations of the actuators.
- Mode controller and Reference signals define switch settings and reference values used by the control laws.
- Wind and Turbulence determines the component of wind and atmosphere turbulence in the aircraft's body axes along with the time derivatives of these values.
- Sensors gathers other sensor characteristics and is used to subtract the initial conditions from the S-function outputs that leave the system Beaver (this is necessary, because the autopilot control laws are based upon deviations from the initial values of S-function outputs while the aircraft model itself uses the full signals).
- Add initial inputs is used to add the initial values of the control inputs to the changes in control surface deflections according the control laws (again: the aircraft model is based upon the full signal; the control laws are based upon deviations from the initial values).

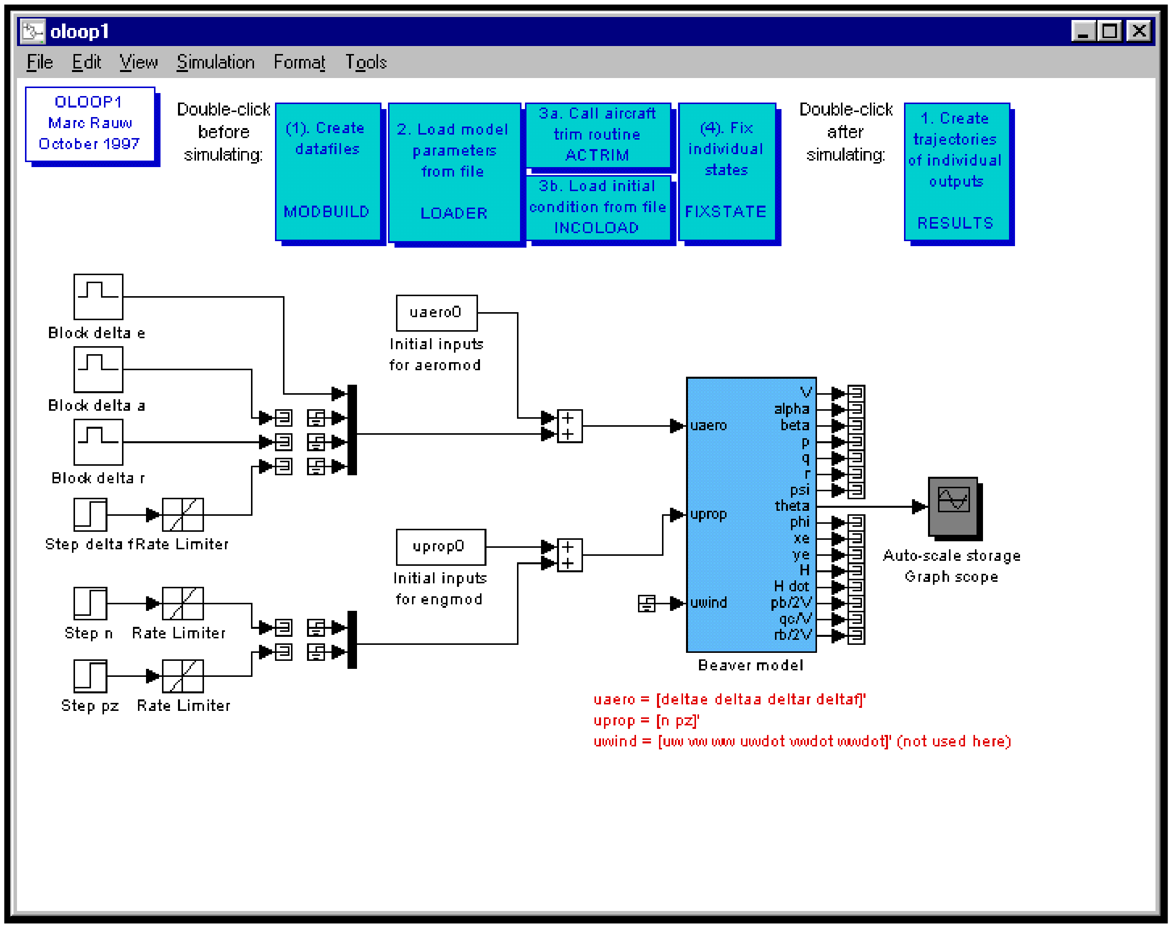


Beaver dynamics and autopilot by Marc Rauw, January 1998 modified by Diego Del Gobbo, March 2000
wind, gusts & turbulence
uwind
[x; Hdot]
uaero
uprop
y dl
Beaver dynamics
Sensors subtract initial conditions
z
z-z0
Actuator & cable dynamics
Ddeltae
Ddeltaa
Ddeltar
n
Add initial inputs
Autothrottle
D_rud ref
D_ail ref
Asymmetrical autopilot modes
computational delay & limiters
D_elv ref
Symmetrical autopilot modes
Mode Controller
Input sequence

Figure 11: Top-Level Schema of the Beaver Simulator

4.3.1 Performing Simulations

The set up for performing a simulation is shown in the Figure 12 .By using this set up we generated various simulations. Before starting the simulation, it is necessary to define the system parameter in the MATLAB workspace. First of all, "Beaver" requires the parameter vector GM1, and the parameter matrices AM, EM, and GM2 to be present in the MATLAB workspace. Next, the initial flight condition must be defined or computed. One can use ACTRIM to determine the steady flight condition or use INCOLOAD to load a flight condition from file. If you are interested in keeping state variables from the aircraft model fixed to their initial value can use FIXSTATE option. After initializing all the variable you can start the simulation. Monitoring of the simulation can be done in the graphics window by its time trajectory or by making the desired plots. After completing the simulation the workspace contains the variables *Time*, *In* and *Out*.



uaero = [deltae deltaa deltar deltaf]'
uprop = [n pz]'
uwind = [uw vw ww uwdot vwdot wwdot]' (not used here)

Figure 12: Block Diagram for Performing Simulation

4.4 Verification Environment for Fault Tolerant requirement

Fault tolerant requirements are specific for the FCS that has a fault tolerant capability. These requirements can be further partitioned into two subsets

- Detection and identification requirements and
- Accommodation requirements.

This partition comes from the different input/output spaces of the two subsets, and from the different roles that they play in the system specifications. Fault detection and identification requirements are strictly related to fault modes. For each fault mode the requirements determine input/output regions of detectability and identifiability. Accommodation requirements address the overall system safety. They have to be satisfied regardless of whether the Flight Control System has fault tolerant capability or not.

4.4.1 Testing environment for accommodation requirement:

After providing the sequence of flight maneuvers to the flight simulator it produces a time framed inputs/outputs file for each trajectory. A file with such an internal data structure as shown in the Figure 13.

Input record 0	Input Record 1	…	Input record i	Input record i+1	…	Input record n
Output record 0	Output record 1		Output record i	Output record i+1		Output record n

Figure 13: File Structure for the Accommodation Testing Data

So in each time slot we stored input record consist of input vector with 12 variable and corresponding to that input record there output record with 89 variable described above.

The framework for accommodation requirement is shown in the Figure 14.

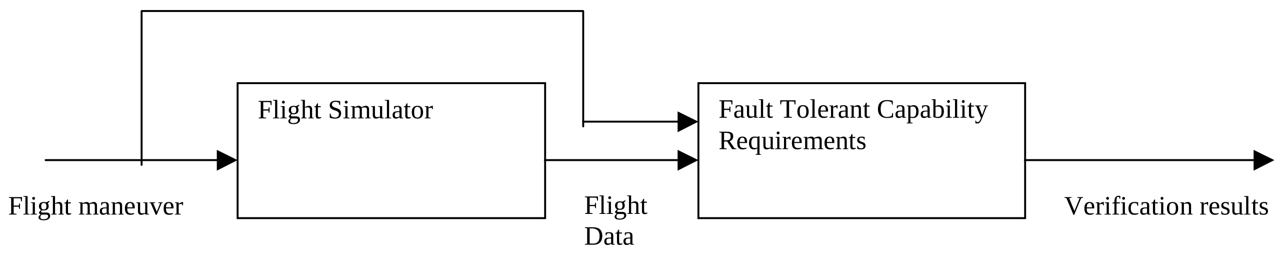


Figure 14 : A Framework for FCS Verification

As we mentioned above that data structure as shown in the Figure 13 is produced after running the simulation. This data structure can be considered a suitable input to the requirement block of the Figure 14.The task of the verification framework is to check whether the requirements are satisfied on the data produced by the simulator with the current values of the flight simulator. The requirements are translated from a relational notation to an executable representation. The use of monotonic operators, most notably of the join operator as structuring device for our specifications plays an important role in the verification step:

4.5 Implementation and Results

In this section we discuss one of the requirement tested by our i.e. coordination in steady banked turns. The prose requirement for that rule is

" The incremental slideslip angle shall not exceed 2 degrees from the trimmed values and lateral acceleration shall not exceed 0.03 g while at steady bank angle up to maneuver bank angle limit reached during normal maneuver with the AFCS engaged"

The corresponding *relational requirement* for that is

Constant terms

Acc_{ϕ} =1:[degree] angle_type;

A_{y_max}= 0.02: [g] accerleration_type;

$\Delta\beta_{max}$ =1:[degree] angle_type;

$RSLF= \{ (m,c) \mid \forall t_1,t_2 :t_1<t_2($

$\forall t :t_1\leq t \leq t_2($

$turb(t)=OFF \Lambda \mid \phi t \mid \leq Acc_{\phi})\Rightarrow$

$\mid \beta(t)-\beta_{trim} \mid \leq \Delta\beta_{max} \Lambda \mid A_y(t) \mid \leq A_{y\_max})\}$

After providing the input condition to the flight simulator the *In* and *Out* matrix are produced for each time slot. The time interval for producing the result was 0.5 sec and the simulation was run for different total times. From the *Out* matrix we were interested in three variables viz. β, ϕ and A_y which were present at positions 3, 37 and 41, respectively. Rest of the variables ,i.e., Acc_{ϕ} , $\Delta\beta_{max}$, A_{y_max} in this specification were constants with the values 1, 1, and 0.02 respectively. In this specification the turbulence switch was in the off position. The value of β_{trim} is -0.0195 (3 variable in xinco).The above relational specification was implemented in C and the output produced by the simulation was fed into these specification to verify the results. From the produced Out matrix first those columns were selected where the value of the column 37 i.e. $\mid \phi t \mid \leq 1$

and among those columns the condition $|\beta(t)-(-0.0195)| \leq 1 \Lambda |A_y(t)| \leq 0.02)\}$ was checked in terms of boolean variable. If this condition holds for all the columns were $|\phi t| \leq 1$ then we can say that the simulation was without any fault and if there were some false values for the condition then the specification is not true against the implementation. In the above implementation of coordination in steady banked turns we were not able to find any false value against the condition that means that the specified value were same as the actual value provided by the flight control system.

Chapter 5

Summary and Future Needs

Software testing is the process of executing a software system to determine whether it matches its specification and executes in its intended environment. Software testing is the least understood part of the development process. Most software testing experts believed testing should be based on the code. This methodology gives the tester an idea of desired behavior, it does not tell the tester whether software works as specified. A new testing methodology that resolve this problem is specification based testing. Despite this, only a small portion of the testing literature deals with specification based testing issues. In testing, informal specification has limited usefulness but the real benefits are gained from formal specification. In our study we used the flight control system requirements which necessitates that rigorous software testing should performed before deployment. In our study relational notation was used to model the requirement of generic flight control system. The advantage of relational approach are that the requirements are partitioned into less complex components. Each component is separately specified with set of relations. Modeling the specification by relations provides many advantages, i.e., unambiguous means of communication between domain experts and specifiers, capturing of requirement in a traceable manner, checking for completeness and minimality. The relations developed for the system specification include quantifiers, most typically universal quantifiers and existential quantifiers.

We used the formal aspect of relational notation in a verification framework where specifications are used as an oracle to test a system implementation. The issue, how to deal with time in the relational specification is also discussed. Value of control parameters that are input to the test oracle representing fault tolerant capability requirements are produced by using Beaver Airplane simulator.

The result of the investigation demonstrated that relational specification can be used as testing oracle to test a system implementation .It was also found that the issue of time interval should be handle differently and carefully with respect to both existential and universal quantifiers.

While dealing with issue of time interval with respect to universal and existential quantifiers, if know the nature of function, it is possible to determine cost functions by Shannon theorem regarding the issue of time interval. Although we use relational specification as testing oracle for some of simple relational specification, more case studies are needed to analyze the role of relational specification as testing oracle.

References

[1] Alagar V. S and Periyasamy K. Specification of Software Systems. Springer Verlag,1998.

[2] Austin, S and G. Parkin. Formal Methods: A Survey, Published by National Physical Laboratory, Teddington, Middlesex, U.K., March 1993.

[3] Atlee J.M and J.Gannon. State Based Model Checking of Event Driven System requirements. IEEE Trans SE 19(1): 24-40,1993.

[4] Beizer.B. " Software Testing Techniques", second edition, Van Nostrand,1990

[5] Carrington. D and P.Stocks. A Tale of Two Paradigms: Formal Methods and Software Testing, Proc Eighth Z User Meeting, Springer-Verlag, pp 51-68,1994.

[6] Constance Heitmeyer and Dino Mandrioli. Formal Methods for Real-Time Computing. John Wiley and Sons. New York,1996.

[7] Dorfman M. Requirement Engineering. Software Requirement Engineering,1997.

[8] Goodenough, J. B., and Gerhart, S.L. Toward a theory of test data selection. IEEE Trans on Software Engineering. 1:156-173,1975.

[9] Gobbo Del D, B. Cukic, S. Easterbrook, M. Napolitano. Fault Detectability Analysis for Requirement Validation of Fault Tolerant Systems. 4th IEEE International Symposium on High Assurance System Engineering, Washington, DC., Nov 1999.

[10] Lowry. M., M. Boyd, D. Kulkarani. Towards a theory for integration of mathematical verification and empirical testing, *13th IEEE International Conference on Automated Software Engineerin*g, 1998.

[11] Lutz .R.R, and R.M.Woodhouse. Bi-directional Analysis for Certification of Safety Critical System, *Proc of International Software Assurance Certification Conference*, Chantilly, VA, 199*8*.

[12] Mili A., Ammari H. and Winikoff. M. Requirement Specifications. A Relational Approach.1999.

[13] Mili. A, B. Cukic, T. Xia, R. Ben Ayed. Combining Fault Avoidance, Fault Removal and Fault Tolerance: An Integrated Model, *14th IEEE International Conference on Automated Software Engineerin*g, Cocoa Beach, FL, Oct. 1999.

[14] Mili. A, Jules Desharnais and F.Mili. On the Construction of Programs from Relational Specification .1997.

[15] MIL-F-9490D. Flight control system design, installation and test of piloted aircraft, general specification for 1975.

[16] Miller. E., G. Symons and D. Steiner. Testing of the future.CrossTalk.1994,vol 7.

[17] NTSB Final Report, Accident No. MIA83IA218, September 1983.

[18] NTSB Final Report, Accident No. CHI39IA354, September 1993.

[19] NTSB Final Report, Accident No. NYC96IA169, August 1996

[20] NTSB Final Report, Accident No. FTW97IA144, March 1997.

[21] Peters D.K and D. L. Parnas. Using Test Oracles Generated from Program Documentation. IEEE Trans. Software Engineering. Vol. 24. No 3.1998

[22] O'Malley T.O, D. J. Richardson and L. K. Dillon. Efficent Specification - based Oracles for Critical Systems. In California Software Symposium.1996.

[23] RTCA DO-178B, Software Considerations in Airborne Systems and Equipment Certification RTCA, Washington DC, 1992.
[24] Richardson. D, O. O'Mally, and C. Title. Approaches to the specification based testing. ACM Sigsoft '89: Third Symp. Software Testing, Analysis and Verification(TAV3) ACM(1989) pp 89-96.
[25] Richardson D.J, S.L.Aha and T.O'Malley.Specification-based test oracle for reactive systems. In Proceedings of Fourteenth International Conference on Software Engineering, pages 105-118, Melbourne, Australia,May 1992.
[26] Rierson. L. K. Using the Software Capability Maturity Model for Certification Projects, Proc of International Software Assurance Certification Conference, Chantilly, VA, 199*8*.
[27] Spadling, N. Certification of Software in Airborne Safety Critical Systems - An Equipment Manufacturer’s Viewpoint, *In Software Certificatio*n. Elsevier Science Publishers, pp 85-93,1988.
[28] Wordsworth. J. B. Software Development with Z - A Practical Approach to Formal Methods in Software Engineering. Addison-Wesley, New York, 1992.
[29] Vouk, M.A , McAllister, D and Tai,K.C. An Experiment Evaluation of the Effectiveness of Random Testing of Fault Tolerant Software, Proc ACM Software Testing Workshop, pp. 74-81,July 1886.
[30] Tjee, R.T.H and J. A. Mulder. Stability and Control Derivatives of the DeHavilland DHC-2 ‘Beaver’ Aircraft. Technical Report LR-556, Delf University of Technology, Faculty of Aerospace Engineering, Delft, The Netherlands, 1988.
[31] Young .M and R.N. Taylor. Combining state concurrency analysis with symbolic execution. IEEE Trans. SE. 14(10):1499-1511, 1988.
[32] Boudriga. N, F. Elloumi and A. Mili. The Lattice of Specifications: Applications to the Specification Methodology, Formal Aspects of Computing, 4:544-571,1992.
[33] Whittakar, J. A. What is Software Testing ? And Why Is It So Hard. IEEE Software17(1):70-79,2000.
[34] Miller, K., L.J.Morell, R.E. Noonan, S.K. Park, D.M. Nicol, B.W. Murrill and J.W. Voas Estimating the Probability of Failure When Testing Reveals no Failures, IEEE Transactions on Software Engineering 18(1): 33-44, 1992.
[35] Hayes, I.J. Specification Case Studies. Prentice Hall, 2nd Edition,1993.
[36] Ghannon, J., P. McMullin, and R.Hamlet. Dataabstraction implementation, specification and testing. ACM Transcation on Programming Languages and Systems, 3(3): 211-223, 1981
[37] Brooks F. No Silver Bullet: Essence and accidents of software engineering. Computer, April,1987.
[38] Stocks A. Applying formal Methods to Software Testing. Phd Dissertation, The University of Queensland, Australia, December 1993.
[39] Murray.L., J. McDonald and P. Strooper. Specification Based Class Testing With Class Bench. Technical Report 98-12, Software Verification Research Center, The University of Queenslan,1998.

Appendix A (Flight Simulator Inputs and Outputs)

The set of input applied to and the output produced by the simulator is depicted in the Figure Appedix1.

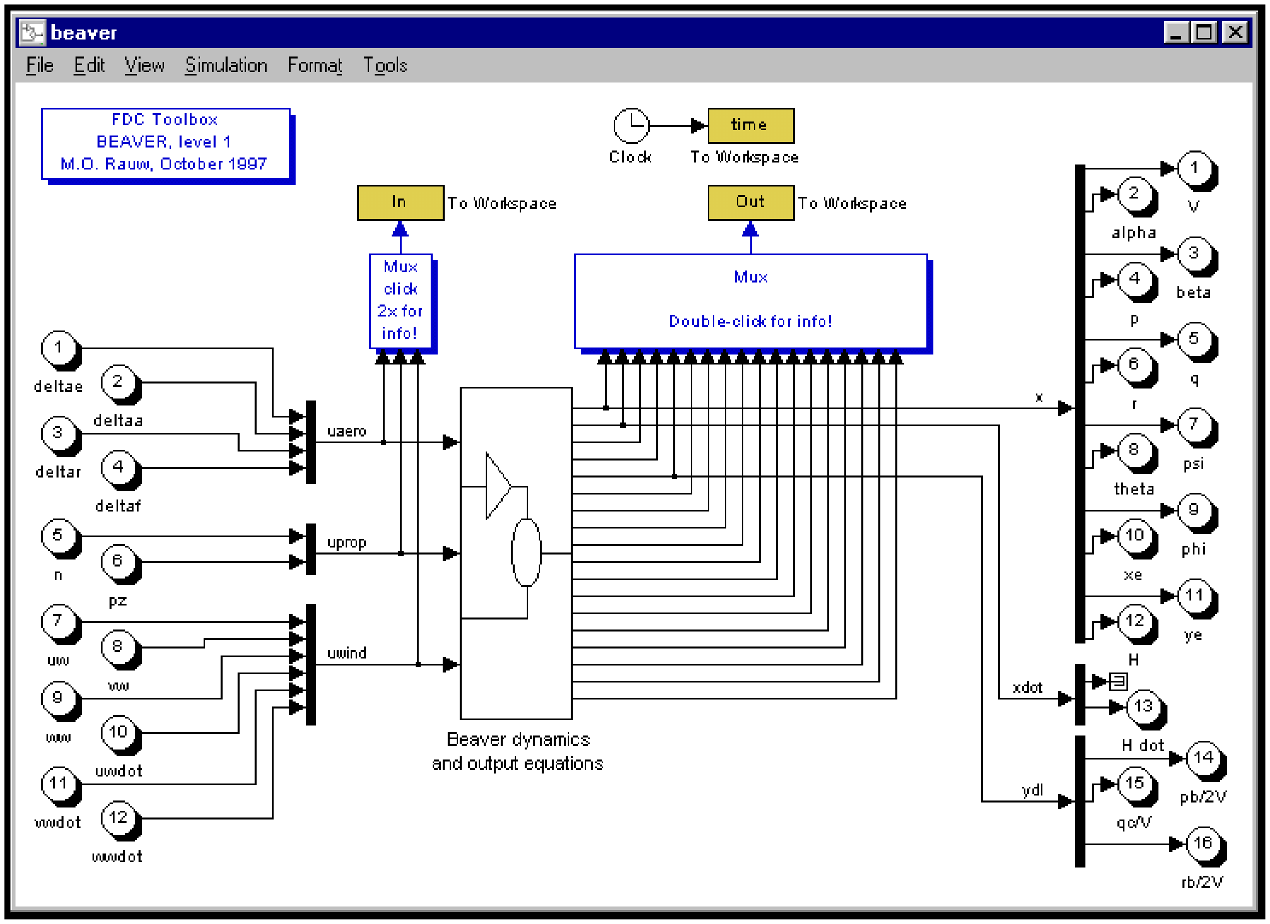


Figure Appendix1: First Level of Graphical Simulink

Simulator Inputs:

There are twelve (scalar) Inport blocks, which means that the system must be accessed with an input vector of length twelve. This input vector (i.e. the input vector to the system Beaver) has been defined as:

```
u     = [uaero'  uprop'  uwind']'
uaero = [deltae deltaa deltar deltaf]'
uprop = [n pz]'
uwind = [uw vw ww uwdot vwdot wwdot]'
where
deltae: elevator deflection [rad]
deltaa: ailerons deflection [rad]
deltar: rudder deflection [rad]
delfaf: flap deflection [rad]
n     : engine speed [RPM]
pz    : manifold pressure ["Hg]
uw    : wind & turbulence velocity along XB-axis [m/s]
vw    : wind & turbulence velocity along YB-axis [m/s]
ww    : wind & turbulence velocity along ZB-axis [m/s]
uwdot : d(uw)/dt [m/s^2]
vwdot : d(vw)/dt [m/s^2]
wwdot : d(ww)/dt [m/s^2]
```

Input variables which are send to the MATLAB workspace. During simulations, the time-trajectories of these input variables are recorded in the matrix *In* in the MATLAB workspace. The matrix *In* contains twelve columns and N rows, where N is the number of time-steps taken during the simulation. The twelve columns correspond with the twelve elements of the inputvector u, so:

In == [u'(t0) ; u'(t1) ; u'(t2) ; ... ; u'(tN)]

The twelve columns of *In* therefore correspond with the variables deltae, deltaa, deltar, deltaf, n, pz, uw, vw, ww, uwdot, vwdot, and wwdot (in this particular order), respectively.

It is important to notice that the number of outputs which are sent to the MATLAB workspace by means of To Workspace blocks is considerably larger than the number of Outport blocks, representing 'S-function outputs' of the system Beaver. The Outport blocks are needed for connecting other systems to the output-side of the system Beaver, e.g., systems with models of sensor dynamics, controllers, etc.

By default, the sixteen outputs which were needed to simulate the 'Beaver' autopilot are connected to Outport blocks. If other output signals are needed by systems which ought to be connected to Beaver, it is necessary to add more Outport blocks to this list of 'S-function outputs'. For instance: if you want to examine control laws which use accelerations as reference signals, these signals must be connected to new Outport blocks in the first level of Beaver.

Currently, it is not possible to send complete vectors through individual Inport and Outport blocks in the first level of a graphical system (where they serve to connect the graphical system to other SIMULINK systems, or to provide access to the system for analytical tools such as trim and linearization routines). Inport and Outport blocks in the first level of graphical SIMULINK(Figure Appendix1) systems only accept scalar signals, which is really a serious limitation of SIMULINK!

<u>S-function outputs</u>

The system Beaver can be treated as a black-box model, which needs to be accessed through its input and output ports only. Beaver currently contains sixteen Outport blocks in the first level of its graphical block-diagram(Figure Appedix1). These sixteen 'S-function outputs' form a subset of the 89 outputs which are sent to the MATLAB workspace (see below). By default, these sixteen outputs are:

States = V, alpha, beta, p, q, r, psi, theta, phi, xe, ye, H ,

Rate of Climb = Hdot

Rotational Speed = pb/2V, qc/V, rb/2V

These variables are needed for simulations of the 'Beaver' autopilot, including the linear state-space models of the dynamics of the control surfaces, steering column/wheel, cables, and actuators. Other sets of 'S-function outputs' can be implemented only by editing the system Beaver according to your own wishes.

During simulations, the time-trajectories of all available output signals are sent to the matrix *Out* in the MATLAB workspace. The columns of this matrix contain the time-trajectories of these outputs, numbered as follows:

Out = [x' xdot' ybvel' yuvw' ydl' ypow' yacc' Caero' Cprop' ...

FMaero' FMprop' Fgrav' Fwind' yatm' yad1' yad2' yad3']'

x =	[V alpha beta p q r psi theta phi xe ye H]'	(1...12)
xdot = dx/dt,	{Vabdot, pqrdot, Eulerdot, xyHdot}	(13...24)
ybvel = [u v w]'	{uvw}	(25...27)
yuvw = [udot vdot wdot]'	{uvwdot}	(28...30)
ydl = [pb/2V qc/V rb/2V]'	{Dimless}	(31...33)
yfp = [gamma fpa chi Phi]'	{Flpath}	(34...37)
ypow = [dpt P]'	{Power}	(38, 39)
yacc = [Ax Ay Az axk ayk azk]'	{Accel}	(40...45)
Caero = [CXa CYa CZa Cla Cma Cna]'	{Aeromod}	(46...51)
Cprop = [CXp CYp CZp Clp Cmp Cnp]'	{Engmod}	(52...57)
FMaero= [Xa Ya Za La Ma Na]'	{FMdims}	(58...63)
FMprop= [Xp Yp Zp Lp Mp Np]'	{FMdims}	(64...69)
Fgrav = [Xgr Ygr Zgr]'	{Gravity}	(70...72)
Fwind = [Xw Yw Zw]'	{Fwind}	(73...75)
yatm = [rho ps T mu g]'	{Atmosph}	(76...80)
yad1 = [a M qdyn]'	{Airdata1}	(81...83)
yad2 = [qc Ve Vc]'	{Airdata2}	(84...86)
yad3 = [Tt Re Rc]'	{Airdata3}	(87...89)

The numbers of the corresponding columns in the output matrix *Out* have been put between round brackets. After finishing a simulation, the time-trajectories, stored in the matrix *Out* can be plotted against the time-axis which is stored in the vector time. For instance, if you want to plot the nth column of *Out*, the plot-command looks like:

plot(time,*Out*(:,n)) where 1 <= n <= 89. The appropriate value of n can be retrieved from the list above. It is important to notice that this list represents the default definition of *Out*, used in the system Beaver. You may wish to add more outputs, or delete unwanted outputs from this list, by adding and/or deleting blocks to/from the system. If you want to plot simulation results by directly using the matrix *Out*, as demonstrated above, you need to know the column numbers of the different outputs. However, if your system uses the same definitions of the matrices In (see the list of input signals) and *Out* as the system Beaver, it is also possible to run RESULTS before plotting the results, in order to get separate time-trajectories of all input and output variables with self-explaining variable names.

<u>Variable used in the Out Matrix</u>

V	: airspeed [m/s]
alpha	: angle of attack [rad] or [deg]
beta	: sideslip angle [rad] or [deg]
p	: roll-rate [rad/s] or [deg/s]
q	: pitch-rate [rad/s] or [deg/s]
r	: yaw-rate [rad/s] or [deg/s]
psi	: yaw-angle [rad] or [deg]
theta	: pitch-angle [rad] or [deg]
phi	: roll-angle [rad] or [deg]

xe	: X-coordinate in Earth-axes [m]
ye	: Y-coordinate in Earth-axes [m]
H	: altitude [m]
Vdot	: time-derivative of airspeed [m/s^2]
alphadot	: time-derivative of alpha [rad/s] or [deg/s]
betadot	: time-derivative of beta [rad/s] or [deg/s]
pdot	: time-derivative of p [rad/s^2] or [deg/s^2]
qdot	: time-derivative of q [rad/s^2] or [deg/s^2]
rdot	: time-derivative of r [rad/s^2] or [deg/s^2]
psidot	: time-derivative of psi [rad/s] or [deg/s]
thetadot	: time-derivative of theta [rad/s] or [deg/s]
phidot	: time-derivative of phi [rad/s] or [deg/s]
xedot	: time-derivative of xe [m/s]
yedot	: time-derivative of ye [m/s]
Hdot	: time-derivative of H [m/s]
u	: component of V along XB-axis [m/s]
v	: component of V along YB-axis [m/s]
w	: component of V along ZB-axis [m/s]
udot	: time-derivative of u [m/s^2]
vdot	: time-derivative of v [m/s^2]
wdot	: time-derivative of w [m/s^2]
pb/2V	: dimensionless roll-rate; b is the wingspan [m]
qc/V	: dimensionless pitch-rate; c is the mean aerodynamic chord [m]

rb/2V	: dimensionless yaw-rate; where b is the wingspan [m]
gamma	: flightpath angle [rad] or [deg]
fpa	: flightpath acceleration [m/s^2]
chi	: azimuth angle [rad] or [deg]
Phi	: bank angle [rad] or [deg]
dpt	: dimensionless pressure increase across propeller [-]
P	: engine power [Nm/s]
Ax	: specific force along XB-axis [g]
Ay	: specific force along YB-axis [g]
Az	: specific force along ZB-axis [g]
axk	: kinematic acceleration along XB-axis [g]
ayk	: kinematic acceleration along YB-axis [g]
azk	: kinematic acceleration along ZB-axis [g]

CXa	: coefficient of aerodynamic force along XB-axis [-]
CYa	: coefficient of aerodynamic force along YB-axis [-]
CZa	: coefficient of aerodynamic force along ZB-axis [-]
Cla	: coefficient of aerodynamic moment around XB-axis [-]
Cma	: coefficient of aerodynamic moment around YB-axis [-]
Cna	: coefficient of aerodynamic moment around ZB-axis [-]
CXp	: coefficient of engine force along XB-axis [-]
CYp	: coefficient of engine force along YB-axis [-]
CZp	: coefficient of engine force along ZB-axis [-]

Clp	: coefficient of engine moment around XB-axis [-]
Cmp	: coefficient of engine moment around YB-axis [-]
Cnp	: coefficient of engine moment around ZB-axis [-]
Xa	: aerodynamic force along XB-axis [N]
Ya	: aerodynamic force along YB-axis [N]
Za	: aerodynamic force along ZB-axis [N]
La	: aerodynamic moment around XB-axis [Nm]
Ma	: aerodynamic moment around YB-axis [Nm]
Na	: aerodynamic moment around ZB-axis [Nm]
Xp	: engine force along XB-axis [N]
Yp	: engine force along YB-axis [N]
Zp	: engine force along ZB-axis [N]
Lp	: engine moment around XB-axis [Nm]
Mp	: engine moment around YB-axis [Nm]
Np	: engine moment around ZB-axis [Nm]
Xgr	: gravity force along XB-axis [N]
Ygr	: gravity force along YB-axis [N]
Zgr	: gravity force along ZB-axis [N]
Xw	: wind force along XB-axis [N]
Yw	: wind force along YB-axis [N]
Zw	: wind force along ZB-axis [N]
rho	: airdensity [kg/m^3]
ps	: static pressure [N/m^2]

T	: temperature [K]
mu	: dynamic viscosity [kg/(m*s)]
g	: acceleration of gravity [m/s^2]
a	: speed of sound [m/s]
M	: Mach number [-]
qdyn	: dynamic pressure [N/m^2]
qc	: impact pressure [N/m^2]
Ve	: equivelent airspeed [m/s]
Vc	: calibrated airspeed [m/s]
Tt	: total temperature [K]
Re	: Reynolds number per unit length [1/m]
Rc	: Reynolds number with respect to mean aerodyn. chord [-]

Appendix B

This section contains both the plain English and relational   requirements. Each requirement has an header that introduces mointored, controlled, constant, quantified and auxiliary quantities used in the specification.

Requirement 1: Attitude Hold (pitch and roll)
 Plain English: Attitude shall be maintained Attitudes in smooth
air with a static accuracy of ±0.5 degree in pitch attitude (with wings level) and _1:0 degree in roll attitude with respect to the reference. RMS attitude deviations shall not exceed 5 degrees in pitch or10 degrees in roll attitude in turbulence. Accuracy requirements shall be achieved and maintained within 5 seconds of mode engagement for a 5 degree attitude disturbance.

where:

- T_s is the settling time interval
- $\sigma = \xi\omega_n$
- ω_n is the natural frequency
- ξ is the damping ratio
- ϵ_{ss} is the maximum steady state error (accuracy)
- A_r is the reference value

From the settling time requirement we find that:

$$\sigma = \frac{1}{T_s} \ln \frac{A_r}{\epsilon_{ss}} = \frac{\ln 5}{5} \qquad \bullet \qquad (2)$$

Damping requirements from section 3.1.2 are specified in terms of response to perturbation of the controlled variable. Here they have been specified in terms of the response after engagement. In fact the dynamics is the same in the two cases, since the roll angle is fed back through a unity loop.

$$M_{RAH} = \tag{3}$$
$$\quad SW_{RAH}() : time\_type \rightarrow Boolean;$$
$$\quad \phi_r : angle\_type;$$
$$\quad turb() : time\_type \rightarrow Boolean;$$
$$C_{RAH} = \tag{4}$$
$$\quad \phi() : time\_type \rightarrow angle\_type;$$
$$Constant\ terms \tag{5}$$
$$\quad \xi_{min} = 0.3 : pureNumber\_type;$$
$$\quad \sigma = \frac{\ln 5}{5} : [Hz] frequency\_type;$$
$$\quad Acc_{\phi} = 1 : [degree] angle\_type;$$
$$\quad RMS_{\phi} = 10 : [degree] angle\_type;$$
$$\quad \epsilon_d = ? : pureNumber\_type;$$
$$Quantified\ terms \tag{6}$$
$$\quad t,\ t_1,\ t_2,\ T_s,\ T_i : time\_type;$$
$$\quad x() : time\_type \rightarrow pureNumber\_type;$$
$$\quad \omega_n : frequency\_type;$$
$$\quad \xi : pureNumber\_type;$$
$$\quad \phi^*() : time\_type \rightarrow angle\_type;$$
$$Auxiliary\ terms \tag{7}$$

$$\delta(f_1(), f_2()) : (f() : time\_type \rightarrow pureNumber\_type) \times$$
$$(f() : time\_type \rightarrow pureNumber\_type) \rightarrow pureNumber\_type;$$
$$RMS(t_1, t_2, f())\ time\_type \times time\_type \times (f() : time\_type \rightarrow f\_type) \rightarrow f\_type$$
$$switch\_on(SW(), t_1) : (f() : time\_type \rightarrow Boolean) \times time\_type \rightarrow Boolean;$$
$$engaged(SW(), t_1, t_2) : (f() : time\_type \rightarrow Boolean) \times time\_type \times time\_type \rightarrow Boolean;$$

$$R_{RAH} =$$
$$\Big\{ (m,c) \big|\ \forall t_1 : switch\_on(SW_{RAH}, t_1)\Big($$
$$\exists T_s : T_s = \frac{1}{\sigma} \ln \frac{|\phi(t_1) - \phi_r|}{Acc_\phi} \Big($$
$$\forall t_2 : \Big(t_2 \geq t_1 + T_s\ \wedge\ engaged(SW_{RAH}, t_1, t_2)\Big)\Big(\ ($$
$$\forall t : t_1 \leq t \leq t_2\ \Big(\ turb(t) = OFF\ \Big) \Rightarrow$$
$$\forall t : t_1 + T_s \leq t \leq t_2\ \Big(\ |\phi(t) - \phi_r| \leq Acc_\phi\ \Big)\Big)\ \wedge\ ($$
$$\forall t : t_1 \leq t \leq t_2\ \Big(\ turb(t) = ON\ \Big) \Rightarrow$$
$$RMS(t_1, t_2, \phi() - \phi_r) \leq RMS_\phi\ \Big)\Big)\Big)\Big)\Big\}\ \bigsqcup$$
$$\Big\{ (m,c) \big|\ \forall t_1 : switch\_on(SW_{RAH}, t_1)\Big($$
$$\exists T_s : T_s = \frac{1}{\sigma} \ln \frac{|\phi(t_1) - \phi_r|}{Acc_\phi} \Big($$
$$\forall t_2 : \Big(t_2 \geq t_1 + T_s\ \wedge\ engaged(SW_{RAH}, t_1, t_2)\Big)\Big($$
$$\forall t : t_1 \leq t \leq t_2\ \Big(\ turb(t) = OFF\ \Big) \Rightarrow$$
$$\exists x(),\ \omega_n,\ \xi,\ \phi^*()\Big($$
$$\omega_n > 0\ \wedge\ \xi_{min} \leq \xi \leq 1\ \wedge\ \phi^*(t) = \frac{\phi(t) - \phi(t_2)}{\phi(t_1) - \phi(t_2)}\ \wedge$$
$$x(t_1) = 1\ \wedge\ \dot{x}(t_1) = \dot{\phi}^*(t_1)\ \wedge$$
$$\forall t : t_1 \leq t \leq t_2\ \Big(\dddot{x}(t) + 2\xi\omega_n\ddot{x}(t) + \omega_n^2\dot{x}(t) = \omega_n^2\Big)\ \wedge$$
$$\delta(t_1, t_2, x(), \phi^*()) < \epsilon_d\ \Big)\Big)\Big)\Big)\Big)\Big\}$$

Requirement 2: Heading Hold
In smooth air, heading shall be maintained within a static accuracy of ±0.5 degree with respect to the reference. In turbulence, RMS deviations shall not exceed 5 degrees in heading at the intensities specified in ????. When heading hold is engaged, the aircraft shall roll towards wings level. The reference heading shall be that heading that exists when the aircraft passes through a roll attitude that is wings level plus or minus a tolerance.

$C_{HH} =$

$\quad \psi() : time\_type \rightarrow angle\_type;$

$Constant\ terms$

$\quad Acc_{\psi} = 0.5 : [degree]angle\_type;$

$\quad RMS_{\psi} = 5 : [degree]angle\_type;$

$Quantified\ terms$

$\quad t,\ t_1,\ t_2,\ T_s : time\_type;$

$Auxiliary\ terms$

$\quad RMS(t_1, t_2, f())\ time\_type \times time\_type \times (f() : time\_type \rightarrow f\_type) \rightarrow f\_type$

$\quad switch\_on(SW(), t_1) : (f() : time\_type \rightarrow Boolean) \times time\_type \rightarrow Boolean;$

$\quad engaged(SW(), t_1, t_2) : (f() : time\_type \rightarrow Boolean) \times time\_type \times time\_type \rightarrow Boolean;$

$R_{HH} =$

$$\Big\{ (m,c) \mid \forall\, t_1, t_2 : \Big(switch\_on(SW_{HH}, t_1)\ \wedge\ engaged(SW_{HH}, t_1, t_2) \Big) \Big(\Big($$

$$\forall\, t : t_1 \le t \le t_2\ \ \Big(turb(t) = OFF \Big) \Rightarrow$$

$$\forall\, t : t_1 + T_s \le t \le t_2 \quad |\psi(t) - \psi(t_1)| \le Acc_{\psi} \Big)\Big)\ \wedge\ \Big($$

$$\forall\, t : t_1 \le t \le t_2\ \ \Big(turb(t) = ON \Big) \Rightarrow$$

$$RMS(t_1, t_2, \psi() - \psi(t_1)) \le RMS_{\psi}\ \Big) \Big) \Big\}$$

Requirement 3: Heading Select
The aircraft shall automatically turn through the smallest angle to any heading selected or pre selected by the pilot and maintain that heading to the tolerances specified for heading hold. The contractor shall determine a bank angle limit which provides a satisfactory turn rate and precludes impending stall. The heading selector shall have 360 degrees control. The aircraft shall not overshoot the selected heading by more than 1.5 degrees with aps up or 2.5 degrees with flaps down. Entry into and exit from the turn shall be smooth and rapid. The roll rate shall not exceed 10 deg/sec and roll acceleration shall not exceed 5 deg/sec/sec.

$M_{HS} =$

$SW_{HS}() : time\_type \rightarrow Boolean;$

$\psi_r : angle\_type;$

$turb() : time\_type \rightarrow Boolean;$

$p() : time\_type \rightarrow angularVelocity\_type;$

$C_{HS} =$

$\psi() : time\_type \rightarrow angle\_type;$

$Constant\ terms$

$Acc_\psi = 0.5 : [degree]angle\_type;$

$RMS_\psi = 5 : [degree]angle\_type;$

$M_\psi = 1.5 : [degree]angle\_type;$

$p_{max} = 10 : [degree/sec]angularVelocity\_type;$

$\dot{p}_{max} = 5 : [degree/sec^2]angularAcceleration\_type;$

$Quantified\ terms$

$t,\ t_1,\ t_2,\ t_a,\ T_s,\ T_t : time\_type;$

Auxiliary terms (19)

$RMS(t_1, t_2, f())\ time\_type \times time\_type \times (f() : time\_type \rightarrow f\_type) \rightarrow f\_type$

$switch\_on(SW(), t_1) : (f() : time\_type \rightarrow Boolean) \times time\_type \rightarrow Boolean;$

$engaged(SW(), t_1, t_2) : (f() : time\_type \rightarrow Boolean) \times time\_type \times time\_type \rightarrow Boolean;$

$R_{HS} =$ (20)

$$
\begin{aligned}
&\Big\{ (m,c) \Big| \,\forall t_1 : switch\_on(SW_{HS}, t_1)\Big(\\
&\quad \exists T_s : T_s = \textit{provided by the contractor!!!}\ \Big(\\
&\qquad \forall t_2 : \Big(t_2 \geq t_1 + T_s \ \wedge\ engaged(SW_{HS}, t_1, t_2)\Big)\Big(\\
&\qquad\quad \forall t : t_1 \leq t \leq t_2 \ \Big(\ |\psi(t) - \psi_r| \leq \pi\ \Big) \ \wedge\ (\\
&\qquad\quad \forall t : t_1 \leq t \leq t_2 \quad turb(t) = OFF \quad \Rightarrow \\
&\qquad\quad \forall t : t_1 + T_s \leq t \leq t_2 \quad |\psi(t) - \psi_r| \leq Acc_\psi \Big)\Big) \ \wedge\ (\\
&\qquad\quad \forall t : t_1 \leq t \leq t_2 \ \Big(\ turb(t) = ON \quad \Rightarrow \\
&\qquad\quad RMS(t_1, t_2, \psi() - \psi_r) \leq RMS_\psi \quad \Big)\Big)\Big)\Big\} \ \bigsqcup \\
&\Big\{ (m,c) \Big| \,\forall t_1 : switch\_on(SW_{HS}, t_1)\Big(\\
&\quad \exists T_s : T_s = \textit{provided by the contractor!!!}\ \Big(\\
&\qquad \forall t_2 : \Big(t_2 \geq t_1 + T_s \ \wedge\ engaged(SW_{HS}, t_1, t_2)\Big)\Big(\\
&\qquad\quad \forall t : t_1 \leq t \leq t_2 \quad turb(t) = OFF \quad \Rightarrow (\\
&\qquad\quad \forall t_a : \Big(\ t_1 \leq t_a \leq t_2 \ \wedge\ \psi(t_a) = \psi_r\ \Big)\Big(\\
&\qquad\qquad \forall t : t_a \leq t \leq t_2 \quad |\psi(t) - \psi_r| \leq M_\psi \Big)\Big)\ ) \ \wedge \\
&\qquad\quad \forall t : t_1 \leq t \leq t_2 \ \Big(\ |p(t)| \leq p_{max} \ \wedge\ |\dot{p}(t)| \leq \dot{p}_{max}\ \Big)\Big)\Big)\Big)\Big\}
\end{aligned}
$$

Requirement 4: Coordination in steady banked turns
The incremental sideslip angle shall not exceed 2 degrees from the trimmed value, and lateral acceleration shall not exceed 0.03g, while at steady bank angle up to the maneuver bank angle limit reached during normal maneuvers with the AFCS engaged.

$$
\begin{aligned}
&C_{SBT} = \\
&\quad \beta() : time\_type \rightarrow angle\_type; \\
&\quad A_y() : time\_type \rightarrow acceleration\_type; \\
&Constant\ terms \\
&\quad Acc_\phi = 1 : [degree] angle\_type; \\
&\quad \phi_{max} =? : angle\_type; \\
&\quad A_{y\_max} = 0.03 : [g] acceleration\_type; \\
&\quad \Delta\beta_{max} = 2 : [degree] angle\_type; \\
&Quantified\ terms \\
&\quad t,\ t_1,\ t_2 : time\_type; \\
&\quad \bar{\phi} : angle\_type; \\
&Auxiliary\ terms \\
&\quad none
\end{aligned}
$$

$$
\begin{aligned}
&R_{SBT} = \\
&\quad \Big\{ (m, c) \Big| \forall t_1, t_2 \ \Big(\\
&\qquad \exists \bar{\phi} : |\bar{\phi}| < \phi_{max} \Big(\\
&\qquad\quad \forall t : t_1 \le t \le t_2 \Big(\\
&\qquad\qquad turb(t) = OFF \ \wedge \ |\phi(t) - \bar{\phi}| \le Acc_\phi \Big)\Big)\Big)\Big(\\
&\qquad \forall t : t_1 \le t \le t_2 \Big(\\
&\qquad\quad |\beta(t) - \beta_{trim}| \le \Delta\beta_{max} \ \wedge \ |A_y(t)| \le A_{y\_max} \Big)\Big)\Big\}
\end{aligned}
$$

Requirement 5: Coordination in straight and level flight
The accuracy while the aircraft is in straight and level flight shall be maintained with an incremental sideslip angle of _1 degree from the trimmed value or a lateral acceleration of _0:02 g at the cg, whichever is lower.

$Constant\ terms$

$$Acc_{\phi} = 1 : [degree]angle\_type;$$

$$A_{y\_max} = 0.02 : [g]acceleration\_type;$$

$$\Delta\beta_{max} = 1 : [degree]angle\_type;$$

$Quantified\ terms$

$$t,\ t_1,\ t_2 : time\_type;$$

$Auxiliary\ terms$

$$none$$

$$R_{SLF} =$$

$$\Big\{ (m,c) \Big| \ \forall t_1, t_2 : t_1 < t_2 \Big(\forall t : t_1 \le t \le t_2 \Big(turb(t) = OFF \ \wedge \ |\phi(t)| \le Acc_{\phi} \Big) \Rightarrow |\beta(t) - \beta_{trim}| \le \Delta\beta_{max} \ \wedge \ |A_y(t)| \le A_{y\_max} \Big) \Big\}$$

Requirement 6: Altitude hold
Engagement of the altitude hold function at rates of climb or descent less than 2000 fpm shall select the existing indicated barometric altitude and control the aircraft to this altitude as a reference. The resulting normal acceleration shall not exceed 0.2g incremental. For engagement at rates above 2000 feet per minute the AFCS shall not cause any unsafe maneuvers. Within the aircraft thrust- drag capability and at steady bank angles, the mode shall provide control accuracies shown in Table ??. These accuracy requirements apply for airspeeds up to Mach 1.0. ... Following engagement or perturbation of this mode at 2000 feet per minute or less, the specified accuracy shall be achieved within 30 seconds.

$M_{ALH} =$

$SW_{ALH}() : time\_type \rightarrow Boolean;$

$\dot{H}() : time\_type \rightarrow climbRate\_type;$

$H() : time\_type \rightarrow altitude\_type;$

$\phi() : time\_type \rightarrow angle\_type;$

$turb() : time\_type \rightarrow Boolean;$

$A_{n\_trim} : acceleration\_type;$

$C_{ALH} =$

$H() : time\_type \rightarrow altitude\_type;$

$A_n() : time\_type \rightarrow acceleration\_type;$

$Constant\ terms$

$T_s = 30 : [feet]time\_type;$

$H_{max} = 30000 : [feet]altitude\_type;$

$\dot{H}_{max} = 2000 : [fpm]climbRate\_type;$

$Acc_{\phi} = 1 : [degree]angle\_type;$

$Acc_{H_0} = 30 : [feet]altitude\_type;$

$Acc_{H_1} = 60 : [feet]altitude\_type;$

$Acc_{H_2} = 90 : [feet]altitude\_type;$

$Acc_{H_1}\% = 0.3\% : percentage\_type;$

$Acc_{H_2\%} = 0.4\% : percentage\_type;$

$\phi_1 = 30 : [degree]angle\_type;$

$\phi_2 = 60 : [degree]angle\_type;$

$\Delta A_{n\_max} = 0.2 : [g]acceleration\_type;$

$$Quantified\ terms \tag{36}$$

$t,\ t_1,\ t_2,\ T_s : time\_type;$

$\bar{\phi} : angle\_type;$

$$Auxiliary\ terms \tag{37}$$

$max(n_1, n_2) : n\_type \times n\_type \rightarrow n\_type;$

$switch\_on(SW(), t_1) : (f() : time\_type \rightarrow Boolean) \times time\_type \rightarrow Boolean;$

$engaged(SW(), t_1, t_2) : (f() : time\_type \rightarrow Boolean) \times time\_type \times time\_type \rightarrow Boolean;$

$$R_{ALH} = \tag{38}$$

$$\Big\{ (m,c) \,|\, \forall t_1 \ \Big(switch\_on(SW_{ALH}, t_1) \ \wedge \ |\dot{H}(t_1)| \leq \dot{H}_{max} \Big)\Big($$

$$\forall t_2 : \Big(t_2 \geq t_1 + T_s \ \wedge \ engaged(SW_{ALH}, t_1, t_2) \Big)\Big(\big($$

$$\forall t : t_1 \leq t \leq t_2 \ \Big(turb(t) = OFF \ \wedge \ H(t_1) \leq H_{max} \Big) \Rightarrow \big($$

$$\forall t : t_1 \leq t \leq t_2 \ \Big(|\phi(t)| \leq Acc_\phi \Big) \ \wedge$$

$$\forall t : t_1 + T_s \leq t \leq t_2 \ \Big(|H(t) - H(t_1)| \leq Acc_{H_0} \Big) \ \vee$$

$$\exists \bar{\phi} : Acc_\phi \leq \bar{\phi} \leq \phi_1 \ \Big($$

$$\forall t : t_1 \leq t \leq t_2 \ \Big(|\phi(t) - \bar{\phi}| \leq Acc_\phi \Big)\Big) \ \wedge$$

$$\forall t : t_1 + T_s \leq t \leq t_2 \ \Big(|H(t) - H(t_1)| \leq max(Acc_{H_1}, Acc_{H_1\%} H(t_1)) \Big) \ \vee$$

$$\exists \bar{\phi} : \phi_1 \leq \bar{\phi} \leq \phi_2 \ \Big($$

$$\forall t : t_1 \leq t \leq t_2 \ \Big(|\phi(t) - \bar{\phi}| \leq Acc_\phi \Big)\Big) \ \wedge$$

$$\forall t : t_1 + T_s \leq t \leq t_2 \quad |H(t) - H(t_1)| \leq max(Acc_{H_2}, Acc_{H_2\%} H(t_1)) \Big)\Big) \ \wedge$$

$$\forall t : t_1 \leq t \leq t_2 \ \Big(|A_n(t) - A_{n\_trim}| \leq \Delta A_{n\_max} \Big) \,))\, \Big)\Big)\Big)\Big\}$$

Function used in these specifications

$switch\_on(SW(), t_1):$

$\quad (f() : time\_type \rightarrow Boolean) \times time\_type \rightarrow Boolean$

$switch\_on(SW(), t_1) ==$

$\quad SW(t_1) = ON \ \wedge \ \exists T_\epsilon : T_\epsilon > 0 \ \left(\forall t : t_1 - T_\epsilon \leq t \leq t_1 \ (SW(t) = OFF) \right)$

$engaged(SW(), t_1, t_2)$

$\quad (f() : time\_type \rightarrow Boolean) \times time\_type \times time\_type \rightarrow Boolean$

$engaged(SW(), t_1, t_2) ==$

$\quad \left(\forall t : t_1 \leq t \leq t_2 \ SW(t) = ON \right)$

$RMS(t_1, t_2, f()) :$

$\quad time\_type \times time\_type \times (f() : time\_type \rightarrow f\_type) \rightarrow f\_type???$

$RMS(t_1, t_2, f()) ==$

$$\sqrt{\frac{1}{(t_2 - t_1)} \int_{t_1}^{t_2} f^2(t)dt}$$

$max(n_1, n_2) :$

$\quad n\_type \times n\_type \rightarrow n\_type???;$

$max(n_1, n_2) ==$

$\delta(f_1(), f_2()) :$

$\quad (f() : time\_type \rightarrow pureNumber\_type) \times$

$\quad (f() : time\_type \rightarrow pureNumber\_type) \rightarrow pureNumber\_type;$

$\delta(f_1(), f_2()) ==$

$\quad to\ be\ defined$