Avionics Software Certification and Regulation

Kyle DeBates
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
debat006@morris.umn.edu

ABSTRACT

As automation was introduced further into the mechanical world, so too has it found its way into aircraft. While this introduces many new possibilities, and immense convenience to the operation of aircraft, the potential for failure in an aircraft can be high without the proper approach to system design. Concerning the relative lack of research into a field so integral to our modern society and commerce, yet prone to such costly mistakes, this paper aims to utilize a conglomerate of sources to establish an understanding of an avionics design framework. This will include how to identify the requirements of said framework and the organization of these requirements into an wholly inclusive architecture. As a result, an explicit structure of model abstraction, utilizing state machines, has been applied to the complexity of avionics systems to define a design methodology that complies with DO-178C avionics design guidelines. Furthermore, referencing specific case studies of avionics design, this paper will acknowledge effective strategies to strengthen the system design, as well as recommend general architectural improvements based on said case studies and appropriate background for the betterment of DO-178C guidelines.

Keywords

Avionics, Software Design Methodology, Requirements Specification, DO-178C, Astrée, Universal Modelling Language

1. INTRODUCTION

When software was initially introduced into the field of aviation, aside from the actual methodology of the time, implementation was fairly simple as automated software components were used rarely and influenced a relatively minor portion of the operation of the aircraft. Today hundreds, if not thousands of software components can form the integral operations of any commercial aircraft's daily operation. This integration can display massive benefits in terms of scaling up the use of planes for more cargo or passengers, as well as helping to lower the failure rate. This scaling however introduces an exponential curve of complexity, as every single part that is integrated into an existing system often must have some form of reference or compliance to other systems already existing in the aircraft. It is expected that in the near future all aviation systems will demand software that will encapsulate functionality from the aircraft's power, to its flight deck, even to the point of being interconnected with the Air Traffic Management on the ground.[1]

One way in which this issue can be mitigated is through the use of extensive documentation in the form of architectural design and model based structures. This is done so that the software can be properly abstracted for modularity and interconnection in code where necessary, allowing for a broader understanding of such complex program functionality. Thus, strict standards regulating this industry have been utilized for decades. These standards are known as the DO-178, a rule set for aviation software and technology. The DO-178 must be constantly updated as avionics technology evolves, causing these standards to grow in complexity with their associated technology. Partially due to a lack of explicit and transparent software design methodology, this industry lacks a generalized, yet exhaustive, software design process. This complicates the job of avionics software developers when attempting to create avionics software that complies with DO-178C standards, for both component and full system software design.

For this reason, this research, along with its sources, hopes to provide more explicit insight into the effective design process of avionic technology. By offering transparent and concise documentation of the methodology necessary for model based design utilized in avionics, this paper aims to allow for more open and accessible information to assist in furthering research and development in the avionics industry. Expanding on various work in the avionics field, this research will primarily elaborate on the avionics software design methodology presented by Paz and Boussaidi. [4]

This paper will address the necessary terminology, abstraction, and industry regulation integral to an understanding of avionics software architecture in Section 2. This will include expanding on concepts of safety and system failure, as well as how these system states are categorized, and finally a brief exploration of DO-178 industry regulations. Section 3 will elaborate specifically on the software design methodology, using a concise articulation of specific safety requirements. This will include how to architecturally structure these requirements into the software classes, as well as properly implementing them through the use of Universal Modeling Language to validate the theoretical models of the avionics component. Finally section 4 will expand on exam-

UMM CSci Senior Seminar Conference, April 2019 Morris, MN.

ples of avionics system testing and how this testing differs from typical consumer software, as well as what benefits are drawn from this difference.

2. BACKGROUND

2.1 DO-178 Avionics Regulations

The DO-178C, released in 2012, is the third and latest edition of an international avionics design standard that was first established as the DO-178 in 1982. Since the inception of DO-178 it has been the flagship in avionics software regulation, used worldwide for its strict and highly explicit methodology that is often required in fields with high potential for catastrophic damage resulting from any particular system failure. Because of the major hazards involved, avionics design often utilizes what is called a standards based approach, focusing on a set of standards or rules to guide design. This is reflected abstractly in the DO-178C, represented by a form of conceptual scrutiny called Claims-Argument-Evidence (CAE). CAEs are generalized throughout the requirements of these guidelines, where claims are made against potential Contributions to Failure Conditions (CFCs), arguments are presented as to the effects of these failures, and evidence is used to weigh the reality of how much a threat this failure really is. The FAA Advisory Circular (AC) stipulates a set of five failure conditions, ranging from Catastrophic to No Effect, where a Catastrophic failure means the plane cannot safely fly or land. [5] This is important to note as it is also stipulated that no unique failure is allowed to produce a Catastrophic failure by itself, by design.

The DO-178C reflects these failure conditions with a system called Design Assurance Levels (DALs), assigning characters A-E to five failure classes based on their potential CFCs, Level A being Catastrophic failures and Level E being No Effect Failures. There is a comprehensive set of 66 objectives laid out in the DO-178C that must be utilized when designing Level A programming, 65 of which must be applied to Level B programming, 57 to Level C, 28 to Level D, and none of which apply to Level E programming. [5] The focus of DALs are to acknowledge failure states, where specific software failures will affect the operation of the aircraft as a whole. The CAE approach is much of the theoretical framework for the design methodology prescribed by the research and discussed in section 3, where the goals of the system prioritize different CFCs threatening system functionality so as to prioritize Level A programming whilst still maintaining the modularity of the system in a fully comprehensive model.

While the goals and intents of these DO-178C guidelines do provide an effective idea of what an avionics software designer must aim to achieve, it does not provide an apparent process or explicit methodology by which to do so. Professor of Computer Science John Rushby specifically acknowledges the issue of avionics lacking an explicit and transparent design methodology by which to navigate the design of such complex systems, that being the fact that a standards based approach to avionics works well, but it is not known to what extent, or why it does work this well at all. [5] While this is currently feasible, any new components introduced into the modern avionic system could potentially provide Level A failures for completely unknown and/or unexpected reasons. Because of this apparent lack of design practice, the methodology in section 3 will display an explicit, model based hierarchy of standards and design practices, which will first have it's terminology explored in the following section.

2.2 Avionics Software Design Practices and Terminology

When developing software for avionics that is properly compliant with DO-178C guidelines, first a set of system requirements must be determined and categorized that are necessary for the system to properly operate. These system requirements fall into different levels of functionality differentiated by each component's CFCs or DALs. System requirements are first defined by their High-Level Requirements (HLRs) that are derived from the CFCs and the System and Safety Requirements Allocated to Software (SRATS), which are established requirements of the software for the functionality to achieve the goal of whatever component or system is automated. It is important to note that SRATS will be the primary goals of the system, for instance an operational landing gear. The HLRs will describe necessary functions to achieve these goals in plain language. These requirements must envelope the entirety of the software's data constraints so that no system operation parameters are unacknowledged, whether it be considering fuel intake or necessary 'trim' of an aileron. In the case study presented by Paz and Boussaidi regarding a Landing Gear Control System (LGCS), it was acknowledged that it took three full iterations of defining SRATS and their necessary HLRs with industry practitioners to properly identify the system constraints and to compartmentalize the functions in a proper safety modularity. [4]

The structure provided by developing the HLRs and the SRATS is hierarchical, thus naturally corresponding with object oriented programming languages. Physical components, such as a fuel gauge, require a software class that can emulate the components required functionality. The software object instantiated from this class has assigned variables that represent the expected variables in that components run-time environment. In the case of a software class 'fuel gauge', this would mean a variable such as 'fuelLevel' would exist to accurately describe the actual fuel level in the components run-time environment. The design of these software classes will be prescribed in section 3.3, where Low-Level Requirements (LLRs) will be explained as the framework for design of the software class' source code.

Relationships that exist between HLRs naturally descend to abstract relationships between software classes. This is done so as to represent actual real-world relationships in the objects established in these classes. The terminology of the DO-178 describes these relationships as 'traces'. These traces allow one class to be validated by another class' own run-time objects, such as a fuel tank class seeing no problem in fuel draining by itself, but the intake class determines this is not possible because the engine is not currently drawing fuel, thus determining a fuel leak exists between the two classes combined data objects. This idea can be extended to the notion of 'bi-directional traces', where two classes rely on data from each other that may predicate a system failure between their combined data and testing. This organization and these specificities are the framework for LLRs, which in-

tend to reflect the software that must be implemented in order to meet the demands outlined by the HLRs and software architecture. This will include the necessary algorithms to operate functions outlined by an HLR, as well as the proper traces between classes necessary to process and validate data in a serial manner, where an example of these requirements and their structuring are outlined in Figure 1.

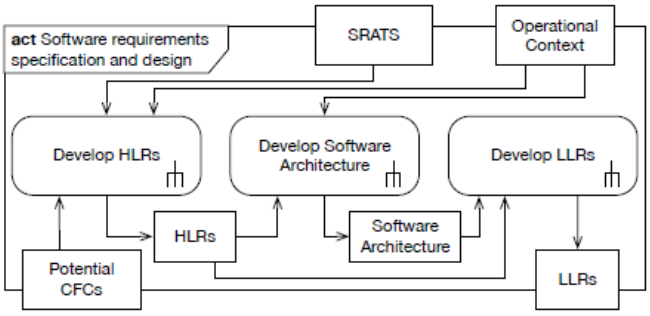

Figure 1: Avionics System as Model [4]

This approach to programming methodology requires software developers to spend large portions of their time identifying fully comprehensive requirements before actually attempting to write code for the system in this 'requirements specification modelling'. Much of coding today follows different consumer based approaches, such as Agile, which focuses on continued iterations of source code to reach a desired final product. This requirements specification modelling, often referred to as 'Big Design Up Front' or the 'Waterfall Model', is common within the engineering field, where an enveloping design must be done from the 'outside in'. That is to say the code should be theoretically proven on paper, or more preferably in a state machine model, to make the system perform to expectations before it is ever iterated in source code. While this method is often considered tedious and thorough, it also produces a highly specialized design through a generalized method.

3. KEY REQUIREMENTS AND PRACTICES CONFERRING DESIGN TO REGULATION

3.1 Developing High Level and Safety Requirements

The process of developing level requirements is partitioned into three major levels concerning overall system requirements, that being to establish HLRs, to establish proper Software Architecture, and finally to establish and develop LLRs. This section will focus on the development of HLR protocol, how it is derived from system requirements, as well as what Critical Failure Conditions are and how these conditions affect HLRs. It is notable that because these requirements are highly predicated on natural language to determine the engineering prerequisites, it is important to be unambiguous, clear and concise. This natural language is used to establish the rule-set to any particular HLR and that HLR's goal to change "controllable variables", or data values the software directly affects such as how much fuel to pump into the fuel injection. These controllable variables are manipulated in response to "monitorable variables", or data values contributed by the environment to sensors that are necessary to operate in said environment, such as the RPM of the engine based on how much fuel we inject. An HLR described this way can be more simply understood as a protocol regarding a specific and necessary system function, like a landing gear extending, and what variables must be accounted for to achieve the successful operation of said functionality. At no point in the process of defining HLRs should the designer lose focus of what controllable variables must exist and be manipulated by the HLR in order to achieve its goal, or it is prone to becoming ambiguous and unclear as to what exact manipulation of controllable variables will be necessary, and more importantly what monitorable variables will be the context of the system.

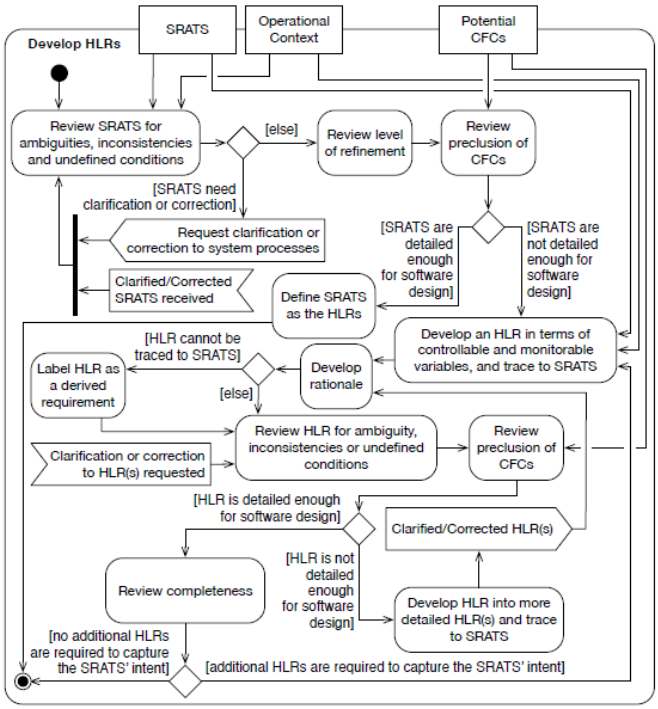

Figure 2: Develop HLRs Activity [4]

In order to begin establishing HLRs, we must identify the SRATS potential Contributions to Failure Conditions (CFCs) as well as manually review the SRATS to confirm they do not include ambiguity, inconsistency, or undefined states. An example of a CFC concerning the previous example of a Landing Gear Control System (LGCS) would be the LGCS creating and communicating actuation commands whilst an LGCS sensor is providing incorrect or invalid data. If, during this process, it is found that an SRATS is specified clearly enough to directly represent software design and not simply set constraints for safety requirements, it is redefined as an HLR. If a trace cannot be properly established between an HLR and any existing SRATs, it is labelled a 'derived HLR'. This is only important in terms of architectural hierarchy, where an example of a derived HLR would be if both the landing gear extending and retracting were defined as the same HLR, and because there is no higher functionality necessary to operate the landing gear, this HLR would not be attributed to an SRAT, where otherwise the opera-

tion of the landing gear could be defined as an SRAT and the retracting and extending as two individual HLRs arbitrarily.

It is at this stage that many bi-directional traces between different HLRs and SRATS will be mapped onto the requirements scheme, which will be determined by the necessary testing to envelope all possible CFCs, and the relevant data and class methods that will define the necessary testing for the CFCs. An example of a bi-directional trace (continuing from the fuel intake example in section 2.1) would be where the fuel intake class verifies that the fuel intake valve is closed, and the fuel tank class verifies the fuel tank is draining, then this collective data would verify an error regarding the integrity of the fuel tank. An example of the HLR activity is abstracted in Figure 2.

3.2 Foundational System Architecture Design

Because the DO-178C requires design models to be inherent to the programming methodology used to design avionic software, the development of the software's architecture and organization is a completely separate activity from determining how the actual functionality will be established. This is done so as to help enable a 'black-box verification' of the overall system behavior, meaning the system is judged based on the inputs and outputs of the software, without necessarily knowing how it achieves these data values. To help accomplish this, design principles that simplify this abstraction are encouraged, such as encapsulation, which prescribes either restricting unnecessary access to certain objects or binding data to their relevant methods to avoid irrelevant methods in other classes. This pattern of modularity is clearly persistent in all layers of system design so as to take a fairly complex system and parse it into a fairly simple, but long, abstraction representing the entirety of system functionality.

One principle way in which this modeling is done is to utilize state machines, an abstract mapping used to identify and outline any systems set of 'potential states' by assigning every 'process' a set of states. Every state has trigger conditions that will begin a transition to another state, and all of these trigger conditions have an associated action that will affect the transition to the following state, which itself also has triggers and actions. Paz and Boussaidi recommend an improvement to this state machine structure, instead assigning 4 characteristics to any state: possible previous states, destination states, transition actions that can change states, and actions triggered when a state is reached. This is done so as to represent an explicit totality of the software, allowing for a much more full and exhaustive understanding of the software's usage and realization to be achieved. [4]

The activity of developing software architecture is performed by designating the software components necessary to achieve the requirements outlined by the HLRs, and any traces or interdependencies that may exist between these components. Every component will will be attributed to a particular HLR, and every component will have data traces defined by expected data interdependencies with other components that hold data relevant to its own operation and/or debugging. Once every requirement outlined by the HLRs is assigned to a component, and every component is attributed to an HLR, hierarchies are established for the necessary software classes for each component. The classes that will be specified in the next activity will all be attributed to particular components, which exist under that particular component's HLRs. This activity is considered complete upon complete establishment of components realizing HLR goals, and class hierarchies realizing component goals.

3.3 Developing Low Level Requirements

Following a thorough assessment and refinement of SRATS and HLRs that properly define the desired working avionics system, as well as a proper outline of the necessary architecture required, it is now time to begin determining the functionality of LLRs that will reflect the intent of every realized class in the software design architecture necessary to perform the requirements specified by the HLRs and SRATS framework. To make this distinction more clear, the SRATS and HLRs previously defined functional requirements necessary to achieve functional goals such as utilizing a landing gear. The architectural design specified what classes would be necessary to execute this code. The LLRs themselves will specify functions necessary for these classes to achieve their assigned functionality.

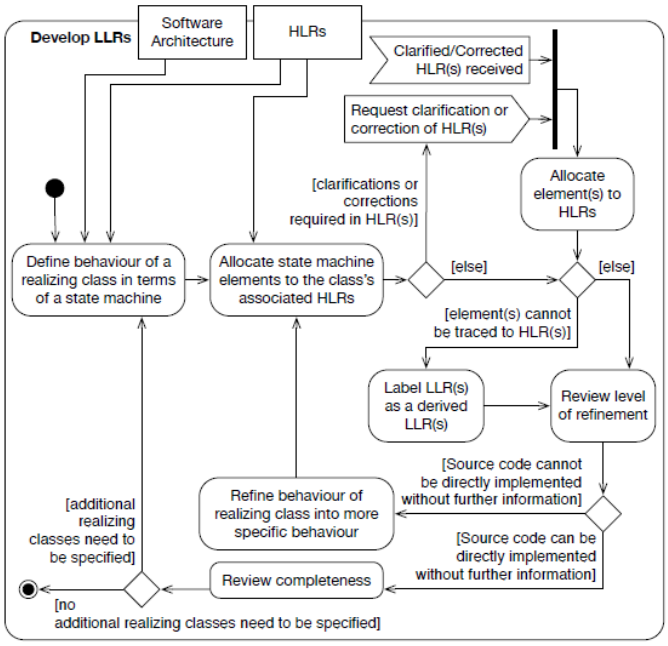

Figure 3: Develop LLRs Activity [4]

The design of the LLRs is highly predicated on the controllable and monitorable variables mentioned prior. These variables will be the data constraints that must be processed and/or manipulated by the LLR processes in order to achieve the desired working system, as outlined by the SRATS, HLRs and architectural classes. Every LLR will be attributed to a particular architectural software class, which itself is attributed to an HLR, as discussed in the previous two sections. If at any point in the design process it is unclear what HLR or architectural class a necessary LLR will pertain to, it becomes necessary to repeat the design HLRs activity so as to clarify necessary functionality and properly assign it to an HLR. If still the LLR still cannot be assigned

to an HLR, it is labelled a 'derived LLR'. For example, an encoded 'system clock', where the software functionality of a clock is necessary for debugging traces. The system clock class itself does not validate being attributed to any particular HLR or software class other than its own class package, thus being a derived LLR. The activity of defining LLRs is considered complete upon the moment that a fully enveloped source code that designates and realizes all necessary classes of the required code without further refinement is established. An example of the develop LLRs activity is given in abstract in Figure 3.

For purposes of black-box verification and abstraction, every software class is alternately defined as a state machine, as discussed in section 3.2. All processes of any particular software class are represented by an arbitrary set of states in the state machine, which are used to simulate all possible and necessary scenarios regarding controllable and monitorable variables that would appear in a run-time environment. State machines are often used in mechanical fields like this with such high standards to establish low failure rates. Particularly when we discuss specific software of such scale, state machines make it relatively easy to properly establish traces between classes without false assumptions regarding this very necessary interconnectivity. The DO-178C stipulates methodology of essential features when designing LLRs being:

> "(i) Layered modelling and hidden decompositions, (ii) Factorization of commonalities or reuse of modelled elements, (iii) partial ordering and concurrent flow of control, (iv) algorithms, (v) time observation and timing constraints, (vi) interruptions in the flow of control and exception handling, (vii) explicit interactions between distinct system parts, (viii) complex trigger conditions and triggered actions, and (ix) flow of data (usage, production, and storage)." [4]

The authors here utilized a UML state machine, utilizing Unified Modeling Language, because of the languages natural efficacy in representing and modelling mechanical LLRs according to the DO-178Cs specific requirements. These requirements represent many of the endeavored constraints for other levels of design mentioned thus far, such as modularity, encapsulation, traceability, and unambiguous, enveloping statements of design intent. A specific example of a UML state machine applied to LLRs, that will be discussed in detail shortly, is displayed in Figure 4, elaborating on the qualities and processes assigned to each state. This abstraction can be organized appropriately by the outlined conditions of the LLRs, such as providing the necessary state for a specific process, as well as explicitly identifying clear CFC's that are considered relevant to the state's being discussed.

Figure 4 outlines the 'Wait For Hydraulic Pressure' LLR of a landing gear operation, where it first refers to HLR-6 which defines the requirement for the hydraulic circuit pressure to be between 30,000 kPa and 35,000 kPa. This HLR accesses its necessary traces and executes its associated LLRs, here being to execute the 'Verify Within Operating Range' LLR to check the sensor values being within this range. If it is within the expected range, the 'Wait for Hydraulic Pressure' LLR will terminate successfully, if not it will timeout according to HLR-12, where the hydraulic system failed to

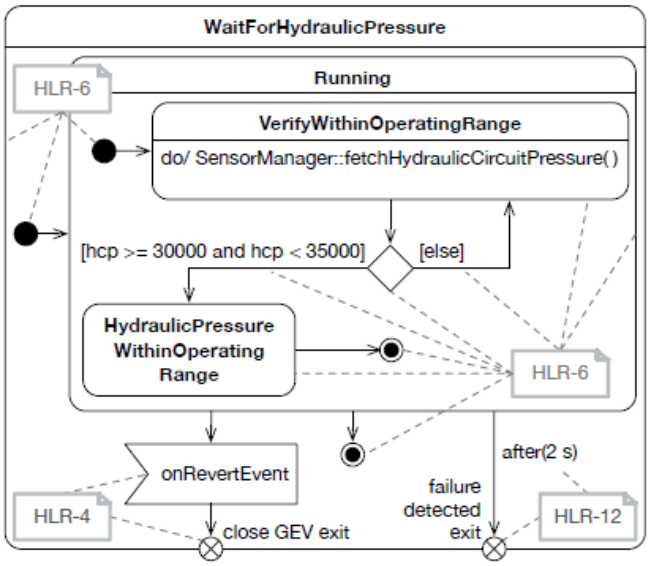

Figure 4: UML State Machine Example State [4]

pressurize. HLR-4 is stipulated as another terminate and revert trace midway through the program operation, not predicated on a particular failure, but for a pilot to revert the action of extending or retracting a landing gear, made under the rationale that a pilot may need to do so. All of these HLRs, their associated LLRs and traces, as well as the overall architecture of the UML state machine representation are described in more explicit detail in Paz's source material. [4]

4. SAFETY ANALYSIS AND EXAMPLES OF TECHNIQUES

4.1 Software Hazard Analysis and Resolution in Design (SHARD)

One of the large benefits of the methodology built thus far is the modularity of the specified architecture. When considering different testing or analysis to ensure the validity of the software once developed focuses on testing the Integrated Modular Avionics (IMA) system as a whole, that is to say to test many of the aforementioned traces that will exist between classes, and also to take advantage of the modularity so as to test individual classes where applicable. Two testing guides emerged from the University of York, namely Software Hazard Analysis (SHARD) and Low-level Interaction Safety Analysis (LISA). SHARD and LISA are both used in tandem, where SHARD monitors unexpected changes in data exchanges between classes and functions, and LISA monitors timed events and system resources for unexpected system errors. [2]

SHARD and LISA will prescribe one of five particular error types in the scenario that one occurs: Omission, where a service is not provided, Commission, where a service is provided when not required, Early or Late, where a service is provided before or after its expected time frame, or Value, where a service receives an unexpected or non-sensical value for the particular operation.[2] Assuming SHARD or LISA

provide the Omission error on a system sensor, we must now begin to refer to CFCs to determine what is likely causing this issue to occur. All mechanisms related to the failing system are tested for three basic failure categories: incorrect functioning of the correct application/IO, incorrect response to an application or IO, and inadvertent function of the IMA system altogether. It is notable that if arguments can be made against the effects of a failure state to disclaim its credibility as an actual system failure, that state can be removed from being considered an error state.

It was found by Conmy and McDermid that four basic error types are prominent, specifically concerning the origin of said errors: functional applications errors such as incorrect calculations, non-functional or behavioral application errors such as a memory violation, computing hardware failures detected by debugging, or external hardware failures such as a sensor no longer working.[2] The two authors prescribe that for every reasonably expectable failure, a set of responses should be defined in case of this failure occurring. An example of this would be to have a backup sensor ready to take the place of a primary sensor, or have the system be prepared to ignore no data from that sensor given a failure so as to avoid improper calculations. [2]

4.2 Astrée Static Analyzer

Another form of testing that largely benefits from the abstract modeling prescribed is Astrée, a static analyzer that is designed to prove the absence of runtime errors, specifically for programs written in C. Being a static analyzer means that Astrée focuses testing on source code inspection, rather than dynamic analysis where a session of system usage is logged and studied after the fact for error. The benefits of this style of testing are broad, where Astrée will always terminate regardless of whether or not the source code itself does. It is fairly efficient at testing, where Astrée takes only one to two hours of testing per 100,000 lines of code, and can properly scale up at this rate to millions of lines of code. Another large advantage of this static analysis is that this large and comprehensive approach to testing can theoretically provide a fully exhaustive testing environment encapsulating the entirety of program execution space.[3]

Astrée prioritizes the idea of traces between programs, mentioned throughout section 3 and earlier, where every trace that maintains communication of dynamic information will have their executions tested within their expected environmental parameters, such as fuel pressure having to be at least zero psi. This prioritization is beneficial as it means that as long as the program's execution is fully tested, there will be no false negatives in the theoretical operation of the system. No system error is ignored in this testing, as opposed to dynamic debugging. Operating in two phases, analysis and verification, Astrée fully computes every potential operation of every class and function, and then checks that none of the processed operations would present a runtime error in a real-time environment. [3] The verification stage is fairly straightforward, where the values culminated from analysis will be checked against expected values. The analysis phases utilizes forms of abstraction to test the actual modelled operation of the software, an example of this being trace partitioning, where the program will test that every single trace embedded in the architecture can send and receive the appropriate data types, regardless of whether this data actually exists within said programs parameters.

This form of testing would arguably be exceptionally useful for the prescribed methodology of this paper, namely if a comprehensive static analyzer was devised in the same manner for use with UML state machine abstractions of code as discussed in section 3.3, especially given the sheer ability of Astrée to effectively reach zero false alarm errors in conjunction with its wide base of theoretical testing. [3] Furthermore, the generalization of this form of testing into high-level specifications languages such as SCADE or SIMULINK and the growing breadth of this theoretical testing, Astrée, or at least revisions based on it, has been used a fair amount in the avionics industry and will likely show greater importance as it is further generalized into other aspects of avionics software testing.

5. CONCLUSION

While the field of avionics continues to enhance the functionality of everyday commercial aircraft, so too does it enhance the need for more comprehensive and explicit documentation that is more transparent to the industry as a whole. The methodology outlined in this research utilizes an exhaustive abstraction model that is compatible with UML state machine representation. It is a recommended adoption to the DO-178C as a formal methodology to follow through on the goals that are stipulated by DO-178C, such as using objective evidence to comply with specific safety protocol. Further recommendations in generalizing source code testing and state machine testing by using software such as SHARD that is adapted to UML state machines could provide lower levels of expected failure rates to further comply with DO-178C standards and provide exceptional documentation on the system in question.

6. REFERENCES

[1] E. Blasch, P. Kostek, P. Paces, and K. Kramer. Summary of avionics technologies. *IEEE Aerospace and Electronic Systems Magazine*, 30:6–11, 09 2015.

[2] P. Conmy and J. McDermid. High level failure analysis for integrated modular avionics. In *Proceedings of the Sixth Australian Workshop on Safety Critical Systems and Software - Volume 3*, SCS '01, pages 13–21, Darlinghurst, Australia, Australia, 2001. Australian Computer Society, Inc.

[3] P. Cousot. Proving the absence of run-time errors in safety-critical avionics code. In *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '07, pages 7–9, New York, NY, USA, 2007. ACM.

[4] A. Paz and G. E. Boussaidi. Building a software requirements specification and design for an avionics system: An experience report. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, pages 1262–1271, New York, NY, USA, 2018. ACM.

[5] J. Rushby. New challenges in certification for aircraft software. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 211–218, New York, NY, USA, 2011. ACM.