Bridging the Gap between Testing and Safety Certification

Aiman Gannous
Department of Computer Science
University of Denver
Denver, CO, USA
aiman.gannous@du.edu

Anneliese Andrews
Department of Computer Science
University of Denver
Denver, CO, USA
andrews@cs.du.edu

Barbra Gallina
IDT, MRTC
Mälardalen University
Västerås, Sweden
barbara.gallina@mdh.se

***Abstract*—DO-178C and its supplement DO-331 provide a set of objectives to be achieved for any development of airborne software systems when model-driven development approaches are in use. Fail-safeMBT is an academic recently proposed model-based approach for testing safety-critical systems. Fail-safeMBT is a potential innovative testing process that needs compelling arguments to be adopted for the development of aeronautical software. In this paper, we reduce the gap between industrial settings and academic settings by adopting the safety case approach and derive substantiation data aimed at arguing Fail-safeMBT compliance with the standards. We explain Fail-safeMBT processes in compliance with software process engineering Meta-Model 2.0, then apply Fail-safeMBT on the Autopilot system. Finally, we link Fail-safeMBT outputs to DO-178/DO-331 process elements, then we derive a substantiation from Fail-safeMBT outputs to support the compelling arguments for achieving certification objectives. Thus, we provide a validation of Fail-safeMBT in the avionic domain.**

TABLE OF CONTENTS

1. INTRODUCTION

Avionic domain related authorities such as the Federal Aviation Administration (FAA) [18] shall recognize that an avionic product is complying with their certification requirements to ensure safety. This recognition of compliance should be documented and reported to all involved stakeholders. However, achieving compliance poses some challenges since stakeholder's knowledge and expertise are diverse. One of the challenges in developing a compliant software is the issue of having conflicting objectives. Blindly seeking compliance with the policies and regulations could obstruct innovation, slow the development process and/or increase the cost [1].

Another challenging issue is the transfer of confidence between the stakeholders in assuring a safety-critical system. One idea is to establish assurance using conformance by acting upon the preparation of software assurance in addition to guiding the use of adequate forms of substantiation. For example, in DO-178C [4], substantiation of any system under consideration for certification must satisfy a set of objectives by conducting test collection, analysis, and reporting reviews.

Standards for certification usually require providing means of justification to eliminate the possibility of misinterpeting problem that software engineers usually stumble upon. For instance, DO-178C provide guidance on preparing a software accomplishment summary as an objective required for certification. When these justifications provided by the developers were structured and detailed cautiously, they might be accepted as means of certification conformance argument either completely or partially [2].

Policies and certificates regulators do not usually explain in detail how the compliance process should be executed rather than just specify what should be executed. This establishes a gap between the developers and the compliance authorities. There are even gaps in using software engineering terms. For example, terms such as black-box testing and white-box testing are widely agreed upon in academia, while in DO-178C certification [4], those terms do not exist. These gaps or disagreements on terms may seem just like a technical issue, but it could contribute to widening the gap even further. Since the steps of certification are objective oriented processes, they concentrate on what should be done without explaining how it should be done. Therefore, using black-box or white-box testing or both methods does not necessary fulfill the certification requirements mandated by the certification authorities because the process of verification is not the same as the process of certification. While certification is about providing evidence to show that all required activities are conducted correctly to satisfy defined objectives, verification is to emphasize on the performance of activities. Bridging the gap between the developers party and the compliance assurance party is receiving much attention lately by software engineering researchers to overcome the challenges. This was the trigger of the rise of software engineering for the compliance topic, which is about encouraging the software engineers to develop their systems in a compliance oriented fashion to eliminate the previous mentioned challenges [1].

For a Safety Critical System (SCS) to be certified in various application domains, developers are required to provide a safety case. According to Bloomfield et al. [23], a safety case is a documented body of evidence that provides a valid argument that the System Under Test (SUT) is safe for a given application in a given environment. Avionic developers shall justify via an implicit safety case that the software performs its intended function with a level of confidence in safety that complies with airworthiness requirements such as DO-178C certificate [3].

In the software engineering literature, many methodologies are available for testing SCSs. In the certification process, the verification methodology itself will be verified. In other words, the certification authority needs to see if the verifica-

tion method used complies with their requirements or not. This raise the question Does the verification methodology meet the certification requirements?. According to Carmen et al. [22], a testing methodology that fulfills all the objectives in a related certification compliance does not exist. Based on these results, we concluded that in order to bridge the gap, we must first examine the testing methodologies prior to deployment. This examination should be based on what objectives can these methodologies meet. We think that assessing a methodology with respect to certification activity will ease and expedite the certification process itself since the developers and the certifiers will jointly know in advance what this methodology will offer in the certification process. This will also motivate the researchers to either enhance their methodologies to meet the certification requirements, combine existing ones, or even innovate new ones in a compliance based structure. Based on this concept we think that the best way to show compliance of a software is by showing the compliance of the development methodology used itself.

Fail-safe Model Based Testing (Fail-safeMBT) is an end-to-end model based testing approach recently proposed by [9] and [10] aimed at increasing robustness by targeting testing for failures and its mitigation behavior. This end-to-end testing process could support the safety claims within the required safety case with the presence of compelling arguments concerning its capability as a verification process evidence. In our previous work [16] we focused on the Plan for Software Aspects of Certification (PSAC). We have used Model-driven safety certificate (MDSafeCer), a model driven based safety certification method recently introduced by [17] to show that Fail-SafeMBT can be partly used as testing-planning related evidence within a safety case. We have focused on specific portions of the DO-178C/D331 documents related to verification process compliance and we have argued about compliance with DO-331- related verification process planning. Software life-cycle data used for compliance substantiation data could be derived from the execution of the verification process. The verification process with respect to DO-178C/DO-331 [5] should detect and report errors that may have been introduced during the development process.

We are aiming at showing how to examine the adequacy of a testing methodology. As a case study, we will discover what Fail-safeMBT could achieve in DO-178C/DO-331 Verification of the Verification Process Results objectives compliance and what it cannot achieve. This will help identify the strengths and the weaknesses of Fail-safeMBT thus urge the need of improving it, or inventing new methodologies regarding software conformance with the standards in safety-critical systems.

Summarizing, the research questions addressed in this paper are:
RQ1. Can Fail-safeMBT provide full compliance if used as a SCS testing methodology in the avionic domain?
RQ2. If Fail-safeMBT does not provide full compliance, what parts of the domain standards can it comply with and what parts does not comply with?

To find answers to these research questions, we will illustrate the processes of Fail-safeMBT using Software and Systems Process Engineering Meta-Model (SPEM 2.0) [12]. Then we will apply Fail-safeMBT to test for failures and its mitigation behavior in the avionic domain (Autopilot System). Then, based on the outputs of the execution of Fail-safeMBT, we will derive compliance substantiation data, i.e. we will derive arguments to be included in the operational safety case showing what objectives related to the Verification of the Verification Process Results are met. We will present the compliance arguments via Goal Structuring Notation (GSN) [14].

This paper is organized as follows: Section 2 gives a background about the related subjects and tools. Section 3 illustrates how we contribute in bridging the gap. Section 4 contains the conclusion.

2. BACKGROUND

DO-178C and DO-331

DO-178C is an industry accepted guidance for developing software for airborne systems and equipment. DO-178C has additional supplements which guide the use of any specific development methods for airborne systems and equipment. For instance, if model-driven engineering approaches are used in the development process, DO-331 must be followed as a guidance. DO-178C and its supplements define a set of objectives with series of processes and expected outcomes. Meeting these objectives guarantees the level of confidence in the correct functioning of the software developed in compliance with airworthiness requirements. According to DO-178C, provision of a series of plans have to be approved by the certification body during the first interaction between the applicant and the certification body itself. The applicant can start the real development if the plans are approved. In the planning process, some of the expected deliverables are: software development plan (SDP), software verification plan (SVP), Plan for Software Aspects of Certification (PSAC) and Software Accomplishment Summary (SAS). The verification process with respect to DO-178C, should detect and report errors that may had been introduced during the development process. Section 6 of DO-178C define the verification process as a process that includes reviews, analysis and tests. More than halve of the DO-178C objectives are verification objectives. These verification objectives are summarized in Tables A3-A7 in DO-178C [6]. DO-178C subsection 6.2 specifies that the person who develop the software should not be the same who perform the testing which provide the satisfactory of the verification independence objective [11]. Another important objective DO-178C promotes that the test cases generation should be requirement-based for both high level and low level requirements. To test if the requirements are met, DO-178C encourages the development of normal and robustness test cases, defining the normal test cases target is to try spotting errors in the system functionality in the normal behavior and conditions, while the robustness test cases are for testing the system in the abnormal and unexpected conditions [6].

Software and Systems Process Engineering Meta-Model 2.0

The Software and Systems Process Engineering Meta-Model Specification Version 2.0 (SPEM 2.0) [12] is a process-oriented OMGs standard for meta-model used as a foundation meta-modeling of software engineering processes. SPEM 2.0 was also employed as the baseline meta-model for the Design Space Exploration process, which finds optimized solutions for the deployment of logical components in terms of tasks or processes (software components) to technical components (hardware resources) with respect to a set of optimization goals and constraints [13].

SPEM 2.0 provide various modelling elements, here we illustrate a description of some of them that are used in this paper

to model Fail-safeMBT processes and Figure 1 represents the symbols associated with each SPEM 2.0 element [12]:

Task: a Method Content Element and a Work Definition that defines work being performed by Roles Definition instances and associated to input and output Work Products. Inputs and outputs are differentiated in mandatory versus optional inputs.
Role: specifies who is/are responsible of performing the task.
Tool: a special Method Content Element that can be used to specify a tools participation in a Task Definition.
Work Product: is a Method Content Element that is used, modified, and produced by Task Definitions.
Guidance: a Describable Element that provides additional information on how the work should be executed.

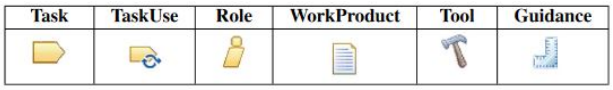

Figure 1. Sample of SPEM 2.0 Symbols.

Goal Structuring Notation (GSN)

GSN is a visual argumentation notation that explicitly characterizes each element of any safety argument and the relationship between them [15]. In any safety critical system, verification of safety assurance with in compliance with standards such as DO-178C, sGSN could be used as a mean of visual representation of the safety case.

GSN consists of the following elements[14]:

Goal: a claim about the system.
Strategy: a description of the inference methodology between a goal and its sub-goal.
Solution: an evidence shows that a particular goal has been achieved.
Context: the domain or scope in which a goal, evidence or strategy is given.
Undeveloped entity: applied to arguments, goals and strategies to indicate absence of existing development.
Undeveloped goal: an intentionally left undeveloped claim in the argument.
Supported by: used to document an evidential relationships by declaring a link between a goal and its substantiated evidence.
InContextOf: indicates a contextual relationship between goal-to-context, goal-to-assumption, goal-to-justification, strategy-to-context, strategy-to-assumption and strategy-to-justification.
Figure 2, illustrates the essential symbols of the GSN notation that used in this research.

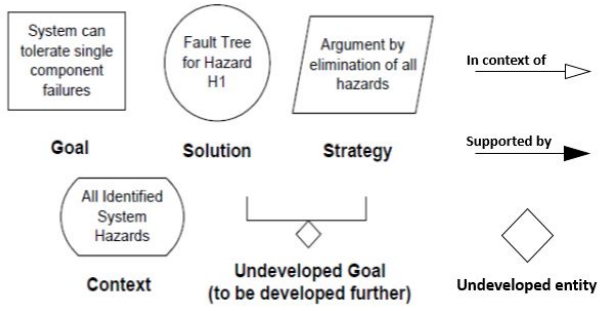

Figure 2. Sample GSN notations

Testing Safety critical Systems

In testing safety critical systems (SCS), hazards identified by any safety analysis technique such as Preliminary Hazard List Analysis (PHL), Failure Mode and Effects Analysis (FMEA) or Fault Tree Analysis (FTA) [19], must be prevented or mitigated by the system under test to ensure safety. In the SCS's we need to perform testing in the presence of failures. Specification-based approaches have some limitations in testing SCS's since they are targeting the desired behavior rather than the undesired one. Sánchez et al. [20], proposed a fault-based methodology to generate test cases for testing safety-critical software. They build a behavior model and used fault trees for identifying the undesirable states in the system. Fault trees then transformed to a state chart entities for compatibility to perform integration with the behavioral model in a flat hierarchy represented in an Extended Finite State Machines (EFSM). Nazier et al.[21] used a similar approach for testing SCS's by transforming Fault Trees events into state chart behavioral elements. They use model checking to verify systems failures. Gario et al.[9] proposed an SCS testing technique. They build a behavioral model using Communicating EFSM's (CEFSM) and a fault models for each possible failure using fault trees. The fault trees models are transformed into CEFSM representation with a compatibility process the integrating them into the behavior model. They apply different graphs coverage criteria to generate test paths to test for failures. Andrews et al.[10] proposed another approach for testing proper failure mitigation of a system based on the safety requirements of the system under test. Their approach uses CEFSM's to model system behavior. Their approach is centered on identifying points of failures in a generated test path using an applicability matrix and coverage criteria. They modeled the mitigation requirements, used them to generated test paths for mitigation and integrate them in the behavioral test paths systematically after each identified point of failure. In this paper we introduce an end-to-end SCS testing approach we call Fail-safe Model Based Testing (Fail-safeMBT). Fail-safeMBT is result of merging Gario et al.[9] approach and Andrews et al.[10] approach. The two approaches share a major activity as they both use CEFSM to build the behavior model for test paths generation. They in fact serve each other in testing SCS's. While Gario et al. approach aims at generating failures, Andrews et al. approach aims at verifying the mitigation for these failures. In the next subsection we will present Fail-safeMBT processes in detail using SPEM 2.0 as a visualization tool and in Bridging the Gap section we will show an application of Fail-safeMBT in the avionics domain, specifically the Autopilot system.

Fail-safeMBT

To make the paper self-contained, here we illustrate the steps of Fail-safeMBT adopted from [9] and [10] as follows:

Building the Behavioral Model (BM) using CEFSM—The first step in the end-to-end process is to build the behavioral model of the system using Communicating Extended Finite State Machines. The input to this process are system components specifications and requirements. The output in this step will be a CEFSM model that depicts the behavior of the system. This step is illustrated using SPEM2.0 in Figure 3.

CEFSM has been used in modeling and testing most types of systems. CEFSM can flatten the orthogonal states of a system without composing the whole system in one state as in state charts. CEFSMs can be defined as a finite set of consistent and completely specified EFSMs along with two disjoint sets

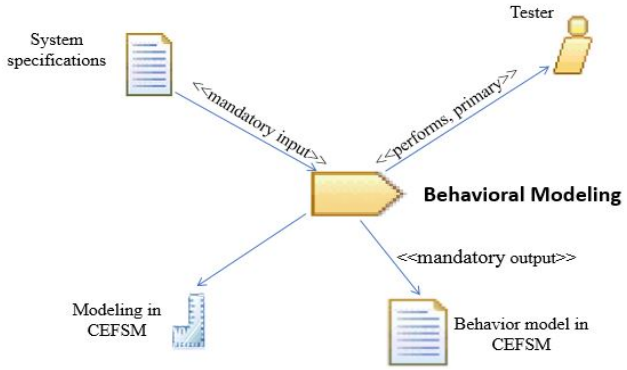

Figure 3. Building the Behavioral Model process in SPEM 2.0

of input and output messages[22]:

CEFSM = (S, s_0, E, P, T, A, M, V, C), such that:

S is a finite set of states,
s_0 is the initial state,
E is a set of events,
P is a set of boolean predicates,
T is a set of transition functions such that:
T: $S \times P \times E \rightarrow S \times A \times M$, where:
A is a set of actions,
M is a set of communicating messages,
V is a set of variables, and
C is the set of input/output communication channel used in this CEFSM.

The function *T* returns a next state, a set of output signals, and action list for each combination of a current state, an input signal, and a predicate. It is defined as:
T(s_i, p_i, get(m_i))/(s_j, A, send(m_{j_1},..., m_{j_k})) where,
s_i is the current state,
s_j is the next state,
p_i is the predicate that must be true in order to execute the transition,
e_i is the event that when combined with a predicate triggers the transition function,
m_{j_1},..., m_{j_k} are the messages.

The communicating message m_i is defined as:
(mId, e_i, mDestination) where,
mId is the message identifier,
mDestination is the CEFSM the message is sent to and,
An event e_i is defined as: *(eId, eOccurrence, eStatus)* where, *eId* is the event type identifier that uniquely identifies it, and *eOccurrence* is set to false as long as the event has not occurred for the first time and to true otherwise, and *eStatus* is set to true each time the event occurs and to false when it no longer applies.

Fault Trees Construction— The second step is to build a Fault Tree (FT) for each possible failure using the safety specifications as inputs. The possible failures are identified in the safety requirments of the system. Safety requirments are used as inputs to this step to prroduce fault trees using Fault Tree Analysis (FTA) as guidence. FTA is a top-down deductive analysis technique used to detect the specific causes of possible hazards [24][25]. The top event in a fault tree is the system hazard. FTA works downward from the top event to determine potential causes of a hazard. It uses boolean logic to represent these combinations of individual faults that can lead to the hazard [25].

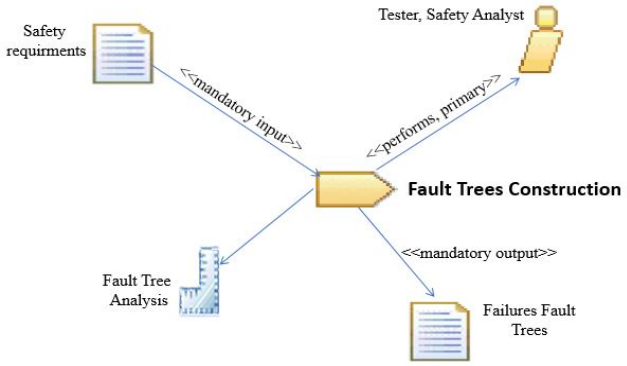

Figure 4. Fault Trees Construction process in SPEM 2.0

A Fault Tree (FT) consists of nodes that represent events, edges that connect nodes to logical gates, and logical gates that represent connectors of events. Every possible major failure in the SUT is represented by separate FT model. This step is illustrated using SPEM2.0 in Figure 4.

Compatibility Transformation—The third step in the process is to establish compatibility between the behavioral model and the fault models to ensure a formal integration. The basic events in fault trees are informally described, therefore the event naming will not match that of the behavioral model. In order to be able to integrate them, we need to make these models compatible first. To establish compatibility between Behavioral and Fault models, we need to put them on the same level of abstraction and events names in both models must match. This step is illustrated using SPEM2.0 in Figure 5.

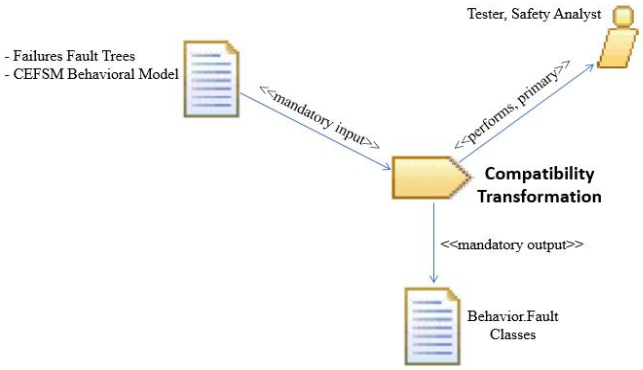

Figure 5. Compatibility Transformation process in SPEM 2.0

The compatibility transformation process takes the Behavioral Model and the FT models as inputs and produces a Behavior.Fault Classes to be used in the next step to execute the transformation. FT entities attributes (each leaf) and the entities attributes of the BM are formalized by creating a the Behavior.Fault class diagram as follows:

1. Identify entities that could contribute to a failure in the behavioral model. An entity could be a state or an event (B_s or B_e).
2. For each such entity, create a *BEntityName* class with behavioral attributes of the state or event.

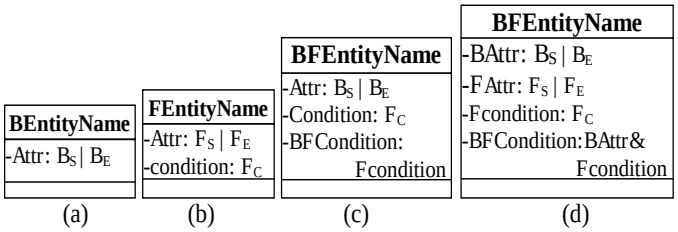

Figure 6. Behavioral and Fault classes combining process

3. Identify leaf node entities from fault trees.
4. For each such entity, create an *FEntityName* class with attributes related to failure and failure condition, a gain they be related to a state or event (F_s or F_e) .
5. Express the failure condition F_C in terms of attributes of *FEntityName*.
6. Combine both *BEntityName* and *FEntityName* in *BFEntityName* by identifying attributes common to both entities such that, if values in *FEntityName* and *BEntityName* are the same, we combine the attributes. Otherwise we create *Battribute* and *Fattribute*.

Figure 6 shows a *BEntityName*, a *FEntityName*, and a *BFEntityName*. The *BEntityName* contains either a state B_S (a state at the behavioral model) or an event B_E (an event at the behavioral model) from the behavioral model that contributes to a failure in the fault model. (These events are carried in the communicating messages from the behavioral to the fault models when these models are integrated.) The *FEntityName* contains a state F_S (a state at the failure model) or an event F_E (an event at the failure model) as described in a leaf node of a FT along with their conditions F_C (if any). The *BFEntityName* contains either a combination of *BEntityName* and *FEntityName* attributes if these attributes are the same as shown in Figure 6 (c) or separate *Battributes* and *Fattributes* are created as shown in Figure 6 (d) when the attributes of *FEntityName* and *FEntityName* are not the same.

The compatibility transformation is an essential step to solve the ambiguity between the events in the behavioral model and fault model.

Fault Trees Transformation—After we made the FT's compatible in step 3, now the forth step in the process is to execute the transformation of the FT's to produce a Gated Extended FSM's ready to be integrated with the BM. The inputs in this step are the BF-Classes from step 3 and the FT's from Step 2 as illustrated using SPEM2.0 in Figure 7. In this subsection we will demonstrate only the transformation rules of the logical gates AND and OR, since they are the two gates to be used later in the application. For the transformation of other type of gates see [9]. Notice that the application of the transformation rules are static, means its a one time implementation and its the same for all domains. Every gate in the FTs is represented as a GCEFSM. This GCEFSM represents a specific part of the failure process. Messages connect it to the behavioral process where failure process and behavioral process interact. The whole model forms a tree-like structure. The ICEFSM consists of a collection of CEFSMs that represent the behavioral model and GCEFSMs (the transformed FT) model. The communication between the behavioral model and FT model is achieved by sending and receiving messages between the models. The behavioral model sends messages that contains events that contribute in the failure to the Fault related GCEFSMs. These GCEFSMs, however, do not send any message back to the behavioral model because they are only used to indicate when the carried events contribute in the root node. Upon receiving those messages, the GCEFSMs at the lower level of the tree sends messages that carry "the event occurred" or "has not occurred" to the upper level GCEFSMs and so on.

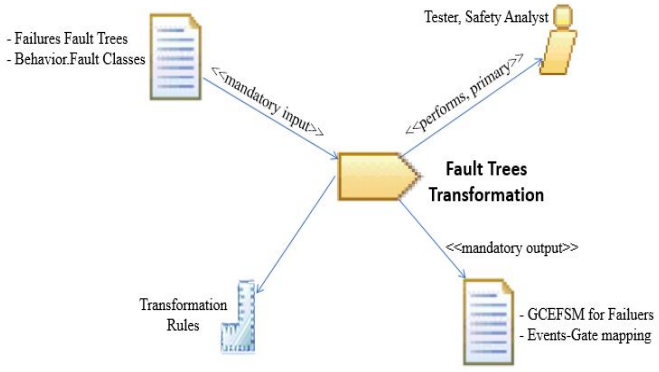

Figure 7. Fault Trees Transformation process in SPEM 2.0

Transformation Rules:— Every GCEFSM is identified by a unique identifier G_i that uniquely identifies it. Each GCEFSM consists of states and transitions that perform the same boolean function as the gates in an FT. The difference is that in the original FT, a gate produces a single output when all the input events satisfy the gate conditions. Otherwise, no output would be produced. However, in the transformed FT, a gate has two kinds of outputs. One output is defined as the "Gate occurred" and the other is defined as "Gate not occurred" such that:

$$m_i = \begin{cases} \text{Gate Occurred} & \text{if } G_i(e_1, e_2, ...e_k) = true, \\ \text{Gate not Occurred} & \text{if } G_i(e_1, e_2, ...e_k) = false \\ & \quad and\ eOccurrence = true \\ & \quad \forall e_i, i = 1\ to\ k \end{cases}$$

For example, an AND gate = true if $G_{AND}(e_1 \cap e_2 ... \cap e_k) = true$. Each structure and behavior of each GCEFSM is predefined and for this matter we will present each gate as follows:

The AND Gate:
When combining some events with an AND gate, the output occurs when all the events occur. Otherwise, no output would occur. An AND gate is represented as shown in Figure 8. It consists of two states and four transitions. State S_0 is the initial state and S_1 is the "gate occurred" state. The transition T_2 will never be taken unless its predicate *NoOfOccurredEvents=TotalNoOfEvents & e_i.eOccurrence =true & e_i.eStatus = true* is true which means all the inputs are received and their status is true. When T_2 is taken the message "gate occurred" is sent to a GCEFSM that is supposed to receive it.

The transition T_0 is as follows:
$T_0 : (S_0, [e_j.eOccurrence = true \& e_j.eStatus = true \& NoOfOccurredEvents < TotalNoOfEvents], get(m_j))/(S_0, update(events), -)$ Where,

1. The event get(m_j) gets input messages from the environment or from another CEFSM. m_i contains an event that could be "gate occurred" or "gate not occurred".
2. update(events) is an action performed upon the executing

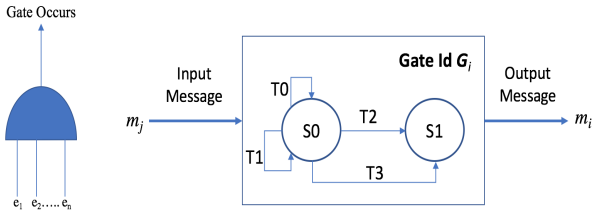

Figure 8. AND Gate representation in CEFSM

of this transition. It updates the number of occurred events and their status based on the last input message received.
3. The predicate "[e_j.eOccurrence = true & e_j.eStatus = true & NoOfOccurredEvents $<$ TotalNoOfEvents]" ensures that the event has occurred and the number of inputs received so far is less than the total number of inputs and the input status is true. Note that "gate not occurred" implies that *eOccurrence=true&eStatus=false*, while "gate occurred" implies that *eOccurrence=true&eStatus=true*.

If all the messages to this GCEFSM are received and all the events have occurred, then the transition T_2 will be taken.

The transition T_2 is as follows:

$T_2 : (S_0, [NoOfPositiveEvents = TotalNoOfEvents \; \& e_j.eOccurrence = true \& e_j.eStatus = true], get(m_j)) /(S_1, update(events), Send(GateOccurred))$

When this transition is taken based on the input and the predicate, it moves to state S_1, increments the number of inputs, and send an output message saying that the gate has occurred.

The transition T_1 is as follows:
T_1: (S_0,[e_j.eOccurrence = true & e_j.eStatus = false], get(m_j))/(S_0, update(events),-)

where "-" means no output produced.

When on state S_0 and the input message implies that the event has changed its status, the transition T_1 is taken. T_1 decrements the number of inputs, and updates the status of the event from occurred to not occurred.

The transition T_3 is as follows:

T_3:(S_1,[e_j.eStatus=false],get(m_j))/(S_0,update(events), Send(Gate not Occurred))

At the state S_1, Transition T_3 is taken when the coming input status is false. When this transition is taken it decrements *NoOfOccurredEvents* and *NoOfPositiveEvents*, updates the status of the input from occurred to not occurred and sends "gate not occurred" message to the receiving gate.

The OR Gate:
The OR gate occurs if at least one event occurs. This gate, as seen in Figure 9, consists of two states and four transitions. When in S_0 and the input message carries an event whose *eOccurrence* and *eStatus* are true (i.e. the event has occurred), T_0 is taken and the OR gate occurs. In state S_1 and if the events in the input messages have not occurred (i.e. their *eStatus* is false) and there was only one input so far, which means this input has changed its status, then a "Gate not occurred" message is sent. Otherwise, no message is sent out of this gate and only update(events) actions take place.

The transitions of the OR Gate are as follows:

$T_0 : (S_0, [e_j.eStatus = true], get(m_j))/(S_1, update(inputs), Send(GateOccurred))$

$T_1 : (S_1, [e_j.eStatus = false \& NoOfPositiveEvents = 0], get(m_j))/(S_1, update(inputs), Send(GatenotOccurred))$

$T_2 : (S_1, [e_j.eStatus = true], get(m_j))/(S_1, update(events), -)$

$T_3 : (S_1, [e_j.eStatus = false], get(m_j))/(S_1, update(events), -)$

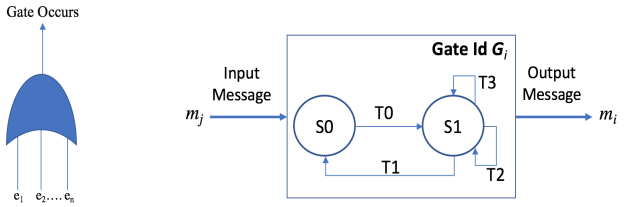

Figure 9. OR Gate representation in CEFSM

Model Integration—Step five in the process is the integration of the BM and the transformed FT's (GCEFSM). The integrated model is the output of this step and it called Integrated Communicating Extended FSM(ICEFSM).

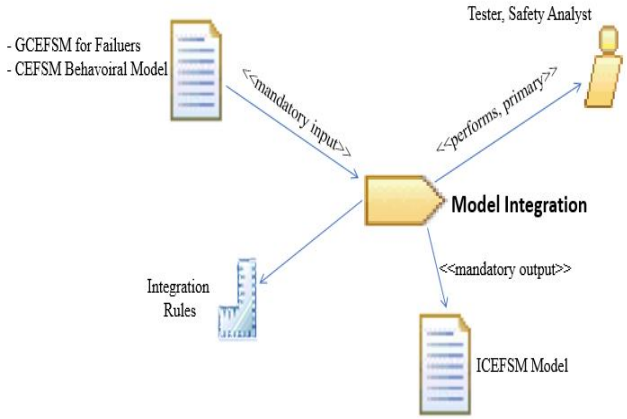

Figure 10. textbfModel Integration process in SPEM 2.0

At that time the *event id* contains the events name and attribute and the receiving gate *id* of that event is not known yet. During the integration of both models, the event name in each message in the behavioral model is looked up in the event-gate table. If the event name and attribute in the behavioral model match those in the event-gate table, the message is modified such that it contains the *event id* e_i and gate *id* G_j and will be modified to (mId, e_j, G_i). This step is SPEM2.0 model is shown in Figure 10.

Test Paths Generation and Failures Mapping—The sixth step in the process is to generate test paths from the BM, the ICEFSM and complete the mapping of the identified failures to the generated test paths to provide traceability. A number of existing test paths generation methods for CEFSMs and

ICEFSM can be used, such as edge-coverage, prime-path coverage, etc.[26]. Test paths generated from ICEFSM will be used to generate test cases to show whether failures can be generated when the behavioral model is in a particular state (reachability) and how to generate failures in various behavioral states. While Test paths generated from CEFSM (BM) will be used to generate test cases to show whether the system behaves as expected in the normal situations. This step is illustrated using SPEM2.0 in Figure 11.

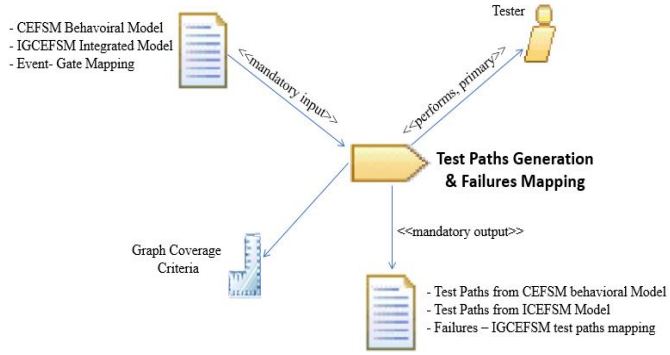

Figure 11. **Test Paths Generation and Failures Mapping process in SPEM 2.0**

Applicability Matrix Generation— The seventh step in the process and the main step in testing for proper mitigation in failing scenarios. At this point, outputs from step six will be used as inputs to determined whether there is a path from a specific behavioral state to a failure or not. Then, we use this information to create the applicability matrix that determines which combination of failure and behavioral state need to be tested for proper mitigation. This step is illustrated using SPEM2.0 in Figure 12.

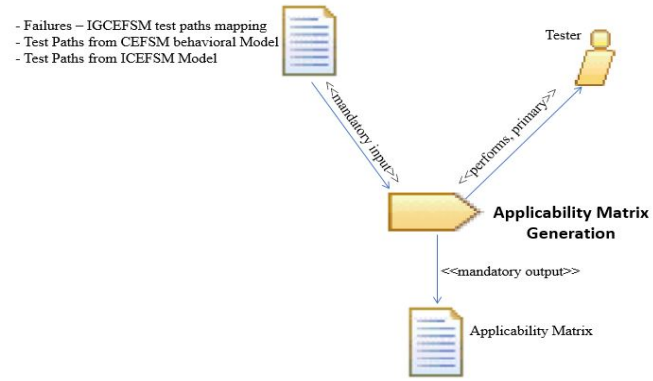

Figure 12. **Applicability Matrix Generation process in SPEM 2.0**

The applicability matrix is a two dimensional array. Each column represents a specific behavioral state s ∈ S and each row is a specific failure type e ($1 \leq e \leq |E|$).

$$A(i,j) = \begin{cases} 1 & \text{if } failure\ type\ j\ can\ occur\ in\ state\ s_i, \\ 0 & \text{if } otherwise \end{cases}$$

The applicability matrix indicates that a specific failure type f_j applies or is relevant to a particular state *s* and "0" depicts that the specific failure type f_j is not applicable in s.

At this point, we know which failures are reachable from which behavioral state s_j via test paths $r \in R$ (i.e. all states in such a test path r receive "1" in the applicability matrix for failure j). We use this information as well as information from the previous steps to generate the applicability matrix.

Mitigation Models Construction— Now in the eighth step, the targeted output are the mitigation models derived from the mitigation requirements and describe mitigation patterns associated with a fault. This step is illustrated using SPEM2.0 in Figure 13.

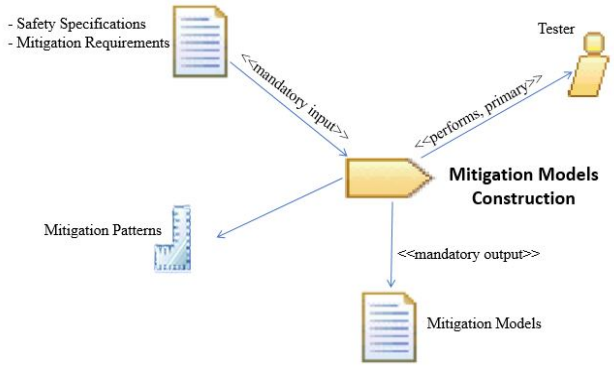

Figure 13. **Mitigation Models Construction process in SPEM 2.0**

Mitigation can take a variety of actions. Mitigation patterns have been defined in [7],[8] as follows:

1. Rollback brings the system back to a previous state before the failure occurred. A mitigation action may occur and the system may stop or proceed to re-execute the remainder of the test.
2. Rollforward mitigates the failure, fixes and proceeds.
3. Try other alternatives deals with decisions about which of several alternatives to pursue.
4. Immediate(partial) fixing when a failure is noted, an action is taken to deal with the problem that caused this failure prior to continuing with the remainder of the test.
5. Deferred (partial) fixing when a failure is noted, an action must be performed to record the situation and deal with the failure either partially or temporarily because handling the failure completely is not possible.
6. Retry when a failure is detected immediately after the execution of the activity causing the problem, an action is performed to solve the failure and then the activity that caused the problem is tried again.
7. Compensate means the system contains enough redundancy to allow a failure to be masked.
8. Go to fail-safe state a system is transferred into a mitigation state to avoid dangerous effects and stops.

These mitigation patterns can be expressed in the form of mitigation models. For example, try other alternatives pattern could be modeled as shown in Figure 14.

Each failure f_i is associated with a corresponding mitigation model MM_i where $i = 1, \ldots, k$. We assume that the models are of the same type as the behavioral model BM (e.g. an EFSM). Graph-based [26], mitigation criteria MC_i can be used to generate mitigation test paths MT_i = $\{mt_{i_1}, \ldots, mt_{i_{k_i}}\}$ for failure f_i. Figure 14 shows an example of a mitigation model of type "Try other alternatives". Assuming the Mitigation Criteria (MC) chosen is

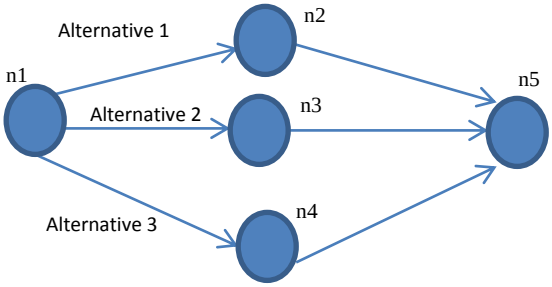

Figure 14. Mitigation Model for "Try Other Alternatives" pattern

"edge coverage", the following three mitigation test paths fulfill MC: MT=$\{mt_1, mt_2, mt_3\}$ where mt_1 =$\{n_1, n_2, n_5\}$, mt_2 =$\{n_1, n_3, n_5\}$, mt_3 =$\{n_1, n_4, n_5\}$

Safety Mitigation Test Paths Generation(SMT):—

Determine Failure Scenarios: — Let the set of failures F be defined as $\{f_1, f_2, f_3, \ldots f_k\}$. A failure is injected into the system by manipulating parameters that indicate to the software under test (SUT) that a particular failure has occurred. This is modeled by selecting a test $bt \in BehavioralTestset(BT)$, determining how far to execute bt (i.e. selecting one of the states in bt) and then inserting a failure injection action at the selected state.

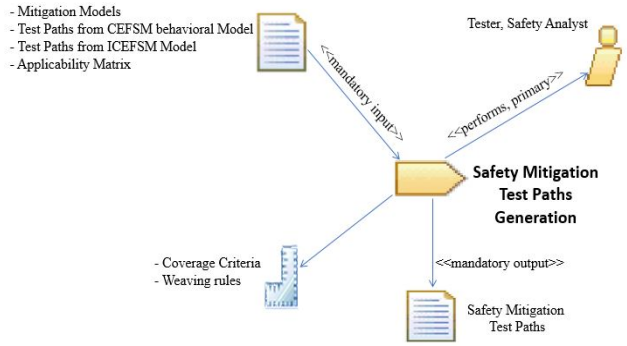

Figure 15. Safety Mitigation Test Paths Generation process in SPEM 2.0

A point of failure is a particular state in a behavioral test path at which the failure is injected. Let CT be the concatenation of test paths in BT. That is $CT = t_1 \circ t_2 \circ t_3 \ldots t_l$. Let $len(t)$ be the number of nodes in t. Then I = len(CT)= $\sum_{i=1}^{l} len(t_i)$. The position of failure is a node in a test t_i (i=1,2,...,l) in the behavioral test suite where the system has failed. While we could describe point of failure as a pair (i,j) where i identifies test t_i and j identifies the j-th node in t_i, it is more convenient to consider the test suite as a series of tests $t_1 \circ t_2 \circ t_3 \ldots t_l$, since it collapses the two dimensions of (i,j) into one, e.g. the third node in test t_2 would now be at position p= len(t_1)+ 3. In general, ($1 \leq p \leq I$). This has the advantage of one fixed length as opposed to variable lengths for each individual test t_i.

We are also selecting failure type e ($1 \leq e \leq |E|$) to apply at the point of failure p. The pairs (p, e) are our potential failure scenarios. The applicability matrix determines which of these are feasible. Let s= node(p) be the state in position p in CT. Then the feasible failure scenarios are PE= $\{(p, e)$ — $1 \leq p \leq I$; $1 \leq e \leq |F|$; A (node(p),e)=1 $\}$. We use coverage criteria to determine test requirements for failure scenarios. Coverage criteria are useful due to systematic algorithmic generation of failure scenarios. Mitigation test criteria describe required coverage. Weaving rules describe how a mitigation test path is woven into the original behavioral test.

Coverage Criteria: — These are based on where in our behavioral test suites failure can occur and need to be tested. In other words, which positions p in the test suite need to be tested with which failure e? The test criteria specify coverage rules for selecting (p,e) pairs from PE. Here we illustrate some of these coverage criteria:
Criteria 1: All combinations, i.e. all positions p, all applicable failure types e (test everything in PE). This is the most expensive. It would require $|PE|$ pairs.
Criteria 2: All unique nodes, all applicable failures. This only requires $\sum_{j=1}^{k} \sum_{i=1}^{|S|}$ (A(i,j)=1) combinations i.e. the number of one entries in the applicability matrix. When some nodes occur many times in a test suite only one needs to be selected by some scheme. This could lead to not testing failure recovery in all tests. A stronger test criterion is to require covering each test as well.

Assuming we have $t \in BT$, p $\in I$, $e \in E$ and mt $\in MT_e$. We now build a safety mitigation test smt $\in SMT$ using this information and the weaving rules $wr_e \in WR$ as follows:

- determine test t that covers position p.
- keep path represented by t until failure position p.
- apply failure of type e (f_e) in p.
- select appropriate $mt \in MT_e$ based on aggregation criteria to guarantee covering all possible failures and using each mt at least once. For example, if MT_e has two mitigation tests mt_1 and mt_2, the aggregation criteria "using each $mt \in MT_e$ at least once" would require constructing two safety mitigation tests, one using mt_1, the other using mt_2. On the other hand if the aggregation criteria was "use e at least one $mt \in MT_e$", we could select either mt_1 or mt_2 and only build one safety mitigation test for failure f_e.
- apply weaving rule wr_e to construct smt.

The result of this step is the full safety mitigation test paths suite SMT. These test paths in SMT are made executable by selecting input values for each input along the path. This step is illustrated using SPEM2.0 in Figure 15.

Autopilot System

Autopilot (AP) is capable of many very time intensive tasks, helping the pilot focus on the overall status of the aircraft and flight. Autopilots can automate tasks, such as maintaining an altitude, climbing or descending to an assigned altitude, turning to and maintaining an assigned heading. AP should be designed to increase safety as well as utility of the aircraft. Safety is enhanced by enabling better situational awareness. Safety can be increased by providing any changes in the modes or any information as fast as possible to enable the pilots to respond fast. One of the most common hazards in using AP is the confusion about the current statues of the AP whether its engaged and doing what it supposed to do or not [18]. In addition to autopilot functions that are engaged by the pilot, some autopilot functions engage and disengage automatically. Pilots sometimes become confused

about whether flight director prompts are being automatically carried out by the autopilot, or left to be manually handled by the pilot. Verification of the autopilot mode and engagement status of the autopilot is a necessary technique for maintaining awareness of who is flying the aircraft. The safety of any flight that uses any type of AP systems can turn to be a liability if the pilots are not aware or confused about the statues of the AP.

3. BRIDGING THE GAP

Fail-safeMBT application on the Autopilot system

We will apply FailSafeMBT on the AP system starting with the system requirements in hand. We model the system components for the purpose of testing the normal system behavior and robustness by testing for safety concerns regarding the AP statues and failures as illustrated in the next subsections.

System requirements:
RQ1: At any time, the pilot can turn the AP switch 'on' and the AP must engage and starts executing a flight plan. A status light should turned and stay 'on' as long as the AP is engaged. Also, an aural recorded message must be played once to confirm the current statues of the AP.

RQ2: At any time, the pilot can turn the AP switch 'off' and the AP must disengage and stop executing any commands. The status light should turned and stay 'off' as long as the AP is disengaged. Also, an aural recorded message must be played once to confirm the current statues of the AP.

System safety requirements:
SFRQ1: If the AP failed to engage, the system controller should perform a retry action. If the retry action failed to engage, the controller must go the fail safe state by engage the Fail-safe Autopilot.

SFRQ2: If the AP failed to disengage, the system controller should take any action to disengage the AP.

SFRQ3: At any time the AP status change from engaged to disengaged and vise versa, an aural recorded message must be played once to confirm the current statues of the AP. If the recorded message failed to play, the system controller must try to play it again.

SFRQ4: At any time the AP status change from engaged to disengaged and vise versa, the statues light must switched on/off accordingly. If the statues light failed to be switched on/off accordingly, the system controller must retry to switch them on/off again.

Building the Behavioral Model

In this step, the system is modeled based on the system requirements and the system components derived from these requirements.
The system consist of five components as follow:
Switch: A toggle that the pilot use to engage or disengage the AP.
Autopilot: Execute the mode task specified by the pilot if engaged.
Lights: Indicator of AP statues.
Aural Engagement Warning: A recorded message that triggered if the AP is engaged.
Aural Disengagement Warning: A recorded message that triggered if the AP is disengaged.

The system is modeled using CEFSM as shown in Figure 16 and the transitions are illustrated in Table 1. Figure 16 depicts the behavioral model of an Autopilot using Communicating Extended Finite State Machine (CEFSM). The model specifies that when the AP switched "on" assuming its armed with one or more flight mode a warning lights are to be turned on when the AP is engaged and an aural warning must be triggered to confirm the status of the AP and keep the pilots aware of who is in control of the plane.

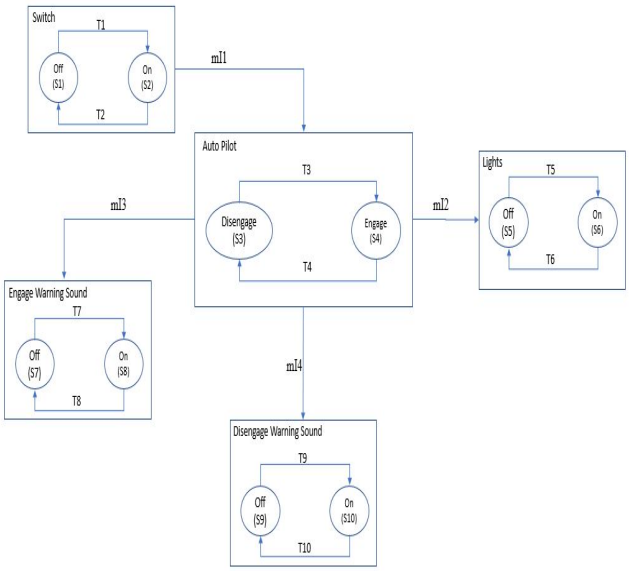

Figure 16. CEFSM model for an Autopilot System

Table 1. Auto pilot awareness CEFSM model transitions

ID	Transition
T1	(Off, [Switch=On],-)/(On, Send mI1 Engage AP)
T2	(On, [Switch=Off],-)/(Off, Send mI1 Disengage AP)
T3	(Disengage, [engage = true], -)/(Engage, Send mI2 Turn Light On, Send mI3 Engage Aural Sound = On)
T4	(Engage, [disengage = true OR Overridden = true], -)/(Disengage, Send mI2 Turn Light Off, Send mI3 Disengage Aural Sound = On)
T5	(On, [Light on = false], -)/(Off, -)
T6	(Off, [Light on = true], -)/(On, -)
T7	(On, [Engage Aural Sound On = false], -)/(Off, -)
T8	(Off, [Engage Aural Sound On = true], -)/(On, -)
T9	(On, [Disengage Aural Sound On = false], -)/(Off, -)
T10	(Off, [Disengage Aural Sound On = true], -)/(On, -)

Fault trees construction

Four omission failures are identified from the safety requirements and classified as Level A hazards according to DO-178C, has to be tested and mitigated properly:

1. Fail to engage the autopilot.
2. Fail to disengage the autopilot.
3. Aural announcement of AP statues fails.
4. Lights that indicate the statues of the AP fails.

This AP system has four fault trees which describe a possible hazardous events. For example, the case of the event Engage is true and the event Switch-Off is true, indicates that the pilot had set the AP to be disengaged and it did not respond, which results in putting the airplane in hazard of possible crash. This is the case when the AP is not executing what is supposed to execute correctly and the pilot cannot take control of the airplane. The fault trees are shown in Figures 17, 18, 19, 20.

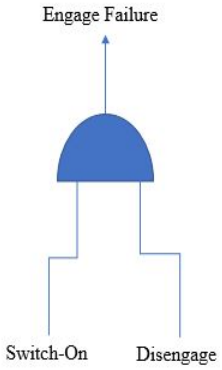

Figure 17. **Engage Failure Fault Tree**

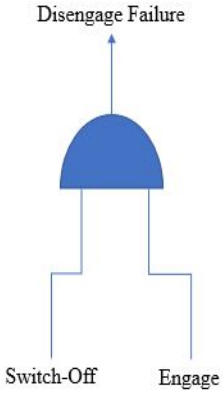

Figure 18. **Disengage Failure Fault Tree**

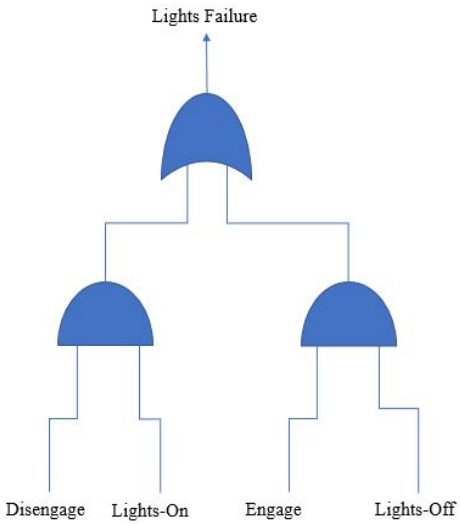

Figure 19. **Lights Failure Fault Tree**

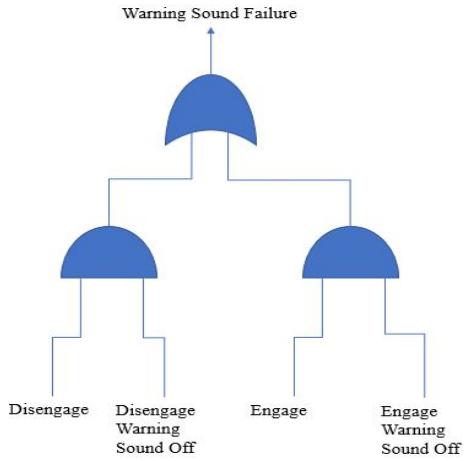

Figure 20. **Warning Sound Failure Fault Tree**

Compatibility Transformation

In this step, the BM and the FMs used as inputs to produce an FTs' that are compatible with the BM. The attributes in each FT are formalized using class diagrams. First, we identify the entities that are subject to fail or cause a failure (leaf nodes). Second, for each one of these entities, we create a behavioral class (BClass) contains the behavioral attributes, and a fault class (FClass) contains failure related attributes. Lastly, we combine both BClass and FClass based on the common attributes. For example, in Figure 17, the first leaf node is Switch-On, this leaf node event is related to the switch behavior therefore we combine their related BClass and FClass into BF-Switch-On class. Notice that some attributes are not considered failures by itself but combined with other events could cause a failure. For example, having the switch on but the AP is not in the engaged state. We continue with the same procedure for all the events in the FTs. The constructed classes are shown in Figures 21, 22, 23, 24, 25. The events in the FT are substituted with the combined attributes from the BF classes that are equivalent to these events. For example, the event Disengage in the FT in Figure 19 is equivalent to BF-Disengage.BFEventCond in the transformed FT in figure 12.

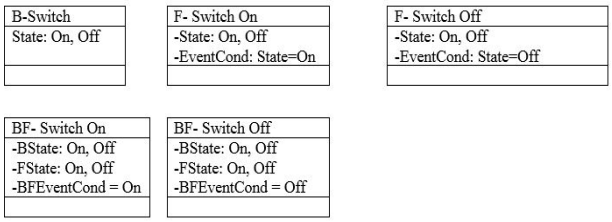

Figure 21. **Switch Class**

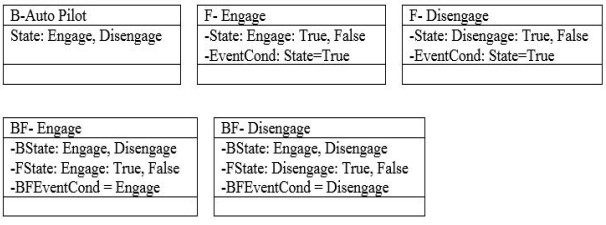

Figure 22. **Autopilot Class**

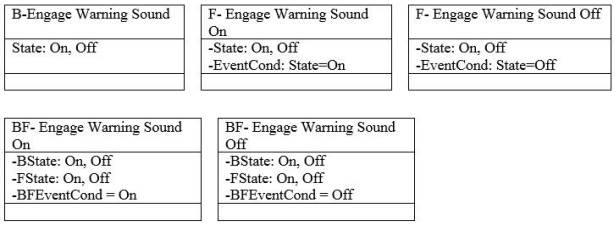

Figure 23. Engage Warning Class

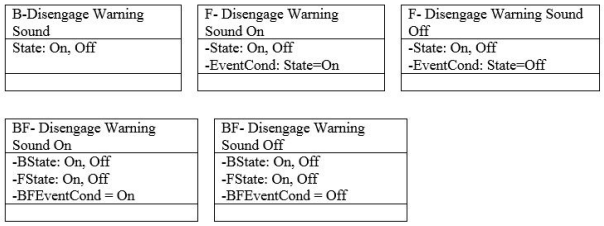

Figure 24. Disengage Warning Class

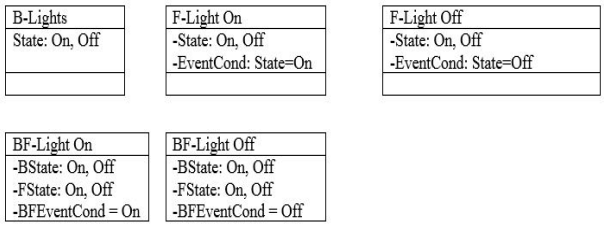

Figure 25. Lights Class

Fault Trees transformation

In this step we transform each FT to a Gate CEFSM (GCEFSM). Starting with the leftmost node in a FT and using the events created in the previous step, we assign an ID to the event, an ID to the gate and insert it in an events to gates mapping table. As illustrated in the Fail-safeMBT in the background section, each type of gate has a formal representation in CEFSM based on transformation rules. The transformed FTs to gates CEFSM (GCEFSM) are represented in figures Figures 26, 27, 28, 29. The event-gate mapping table after the all FTs are transformed is shown in Table 2.

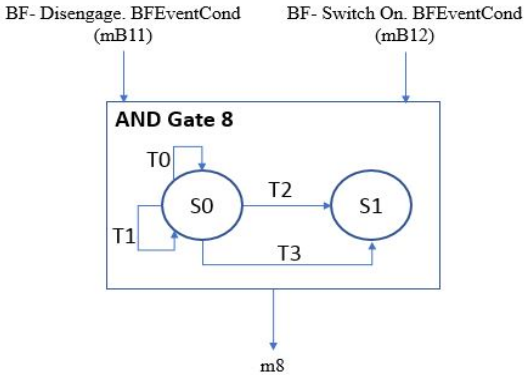

Figure 26. GCEFSM for Engage Failure

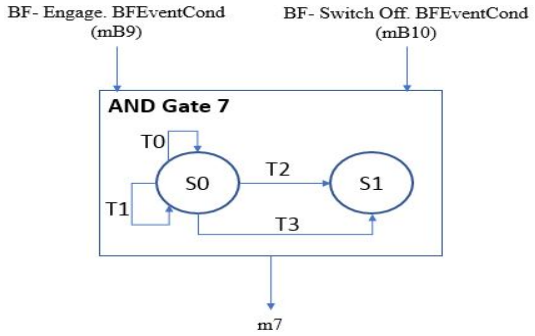

Figure 27. GCEFSM for Disengage Failure

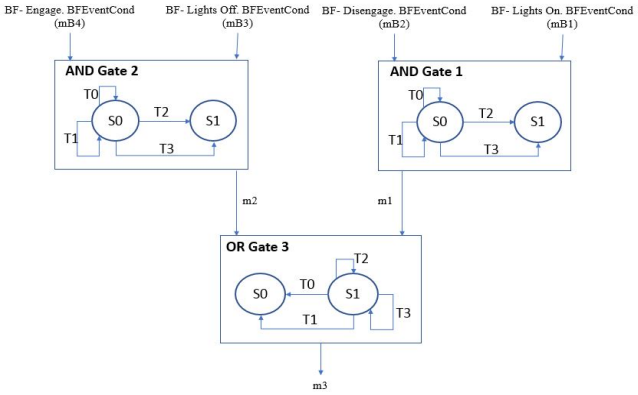

Figure 28. GCEFSM for Lights Failure

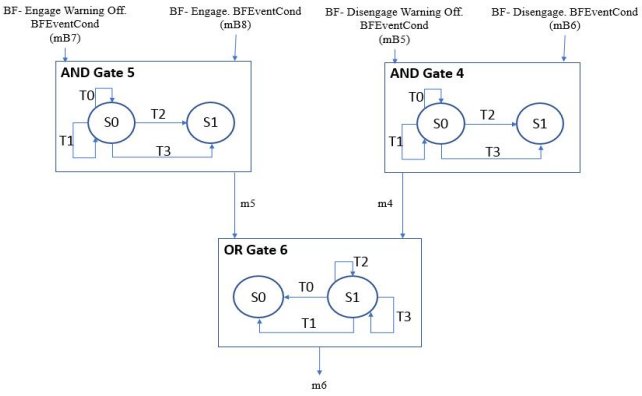

Figure 29. GCEFSM for Warning Sound Failure

Table 2. Events-Gate mapping

Event Name & Attribute	Event ID	Gate ID
BF- Switch On.BFEventCond	eB12	G8
BF- Switch Off.BFEventCond	eB10	G7
BF- Engage.BFEventCond	eB9, eB4,	G7, G2
BF- Disengage.BFEventCond	eB11, eB3,	G8, G2
BF- Lights On.BFEventCond	eB1	G1
BF- Lights Off.BFEventCond	eB3	G2
BF- Engage Warning Sound Off.BFEventCond	eB7	G5
BF- Disengage Warning Sound Off.BFEventCond	eB5	G4

Model Integration

After all the fault trees are transformed into GCEFSMs, we start integrating them into the behavioral model. At this point, every message in the BM contains an event name that is related to an event in one of the leaf nodes of the fault tree. We check the class diagram and the Event-Gate mapping table to find the event ID and the gate ID for the event. These event IDs and gate IDs are inserted into the message at the BM. For example, the event Lights OFF is represented in the class diagram as BFLights-Off.BFEventCond. This event is looked up in the event-gate table to obtain its event ID (eB3) and the gate ID (G2) the message is sent to. The message in the BM is modified as (mB3, eB3, G2). This procedure continues till all the messages in the BM are linked to the FM. Figure 30 illustrates the AP transformed into an ICEFSM model. The integrated model shown in Figure 30 forms a graph to which suitable coverage criteria can be applied to generate test paths. The FT gates that are directly connected to the behavioral model receive messages from the behavioral model and act accordingly. The messages m1 to m8 represent the global transitions between the GCEFSMs for the FT part, while mI1 to mI4 represent the messages between the components of the behavioral model and mB1to mB12 represent the communicating messages between the BM and FM.

Generate test paths and Failures mapping

In this step, we impose an edge coverage criteria on the ICEFSM model in Figure 30, to obtain the test paths shown in Table 4. We also impose an edge coverage criteria on the CEFSM model in Figure 16, to obtain the test paths shown in Table 3. By using reachability analysis, we find that these paths are feasible since there are no conflicts between predicates in transitions. Test paths generated from the integrated model (ICEFSM), could be used for generating test cases for robustness testing, while test paths generated from the Behavioral model (CEFSM model), could be used for Normal test cases generation. Since Fail-safeMBT is built based on the system normal and safety requirements, we claim that these test paths generated from CEFSM and ICEFSM models can be used to generate normal and robustness test cases. Hence, satisfy the Requirements-based testing objective described in DO-178C paragraph 6.4.2. Also, in this step we update the events to gates mapping shown in Table 2 to a Failure to test paths mapping to provide traceability feature. We claim that this mapping illustrated in Table 5 will provide Test cases to Software requirements traceability, which is another requirement for DO-178 certification as described in Section 6.5 of DO-178.

Table 3. Test paths from CEFSM model using edge coverage criteria

Test Path	Sequence
1	S1 $\xrightarrow{T_1}$ S2 $\xrightarrow{M_{I1}}$ S4 $\xrightarrow{M_{I2}}$ S6
2	S3 $\xrightarrow{T_3}$ S4 $\xrightarrow{M_{I3}}$ S8 $\xrightarrow{T_8}$ S7
3	S1 $\xrightarrow{T_1}$ S2 $\xrightarrow{T_2}$ S1 $\xrightarrow{M_{I1}}$ S3 $\xrightarrow{T_3}$ S4 $\xrightarrow{T_4}$ S3 $\xrightarrow{M_{I4}}$ S9
4	S3 $\xrightarrow{T_3}$ S4 $\xrightarrow{T_4}$ S3 $\xrightarrow{M_{I2}}$ S5
5	S5 $\xrightarrow{T_5}$ S6 $\xrightarrow{T_6}$ S5
6	S9 $\xrightarrow{T_9}$ S10 $\xrightarrow{T_{10}}$ S9
7	S7 $\xrightarrow{T_7}$ S8 $\xrightarrow{T_8}$ S7

Table 4. Test paths from ICEFSM model using edge coverage criteria

Test Path	Sequence
1	S1 $\xrightarrow{T_1}$ S2 $\xrightarrow{M_{I1}}$ S4 $\xrightarrow{M_{I2}}$ S6 $\xrightarrow{M_{B3}}$ [2] $\xrightarrow{M_2}$ [3] $\xrightarrow{M_3}$
2	S2 $\xrightarrow{T_2}$ S1 $\xrightarrow{M_{I1}}$ S3 $\xrightarrow{M_{I2}}$ S5 $\xrightarrow{T_5}$ S6 $\xrightarrow{T_6}$ S5 $\xrightarrow{M_{B1}}$ [1] $\xrightarrow{M_1}$ [3] $\xrightarrow{M_3}$
3	S3 $\xrightarrow{T_3}$ S4 $\xrightarrow{M_{B2}}$ [1] $\xrightarrow{M_1}$ [3] $\xrightarrow{M_3}$
4	S3 $\xrightarrow{T_3}$ S4 $\xrightarrow{M_{B4}}$ [2] $\xrightarrow{M_2}$ [3] $\xrightarrow{M_3}$
5	S3 $\xrightarrow{T_3}$ S4 $\xrightarrow{M_{I3}}$ S8 $\xrightarrow{T_8}$ S7 $\xrightarrow{T_7}$ S8 $\xrightarrow{T_8}$ S7 $\xrightarrow{M_{B7}}$ [4] $\xrightarrow{M_4}$ [6] $\xrightarrow{M_6}$
6	S3 $\xrightarrow{M_{I4}}$ S10 $\xrightarrow{T_{10}}$ S9 $\xrightarrow{T_9}$ S10 $\xrightarrow{T_{10}}$ S9 $\xrightarrow{M_{B5}}$ [5] $\xrightarrow{M_5}$ [6] $\xrightarrow{M_6}$
7	S3 $\xrightarrow{T_3}$ S4 $\xrightarrow{M_{B8}}$ [4] $\xrightarrow{M_4}$ [6] $\xrightarrow{M_6}$
8	S3 $\xrightarrow{M_{B6}}$ [5] $\xrightarrow{M_5}$ [6] $\xrightarrow{M_6}$
9	S3 $\xrightarrow{T_3}$ S4 $\xrightarrow{M_{B9}}$ [7] $\xrightarrow{M_7}$
10	S3 $\xrightarrow{T_3}$ S4 $\xrightarrow{T_4}$ S3 $\xrightarrow{M_{B11}}$ [8] $\xrightarrow{M_8}$
11	S1 $\xrightarrow{T_1}$ S2 $\xrightarrow{T_2}$ S1 $\xrightarrow{M_{B10}}$ [7] $\xrightarrow{M_7}$
12	S1 $\xrightarrow{T_1}$ S2 $\xrightarrow{M_{B12}}$ [8] $\xrightarrow{M_8}$

Applicability Matrix generation

The applicability matrix is built based on the information in table Table 5 and the test paths generated from the CEFSM and ICEFSM models. The applicability matrix in Table 6 is basically constructed on the concept of using '1' if the state in the Behavioral Model is a possible contributor to a failure and '0' otherwise. For example, a Lights-On (s6) can fail (f4), hence AM(f4,s6) = 1, but Disengagement Aural Warning failure (f3) is not applicable at state 6.

Table 6. Applicability Matrix

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
F1	1	1	1	1	0	0	0	0	0	0
F2	1	1	1	1	0	0	0	0	0	0
F3	1	1	1	1	0	0	1	1	1	1
F4	1	1	1	1	1	1	0	0	0	0

Mitigation Models

Mitigation requirements are summarized in Table 7. Accordingly, mitigation models are designed as illustrated in Figures 31, 32, 33 and 34, for failures f1, f2, f3 and f4 respectively.

Table 7. Mitigation requirements

Mi.ID	Failure	Action	Mi.Model
MM1	F1	Retry/Use Fail Safe AP	Figure 31
MM2	F2	Fix and Proceed	Figure 32
MM3	F3	Fix and Proceed	Figure 33
MM4	F4	Fix and Proceed	Figure 34

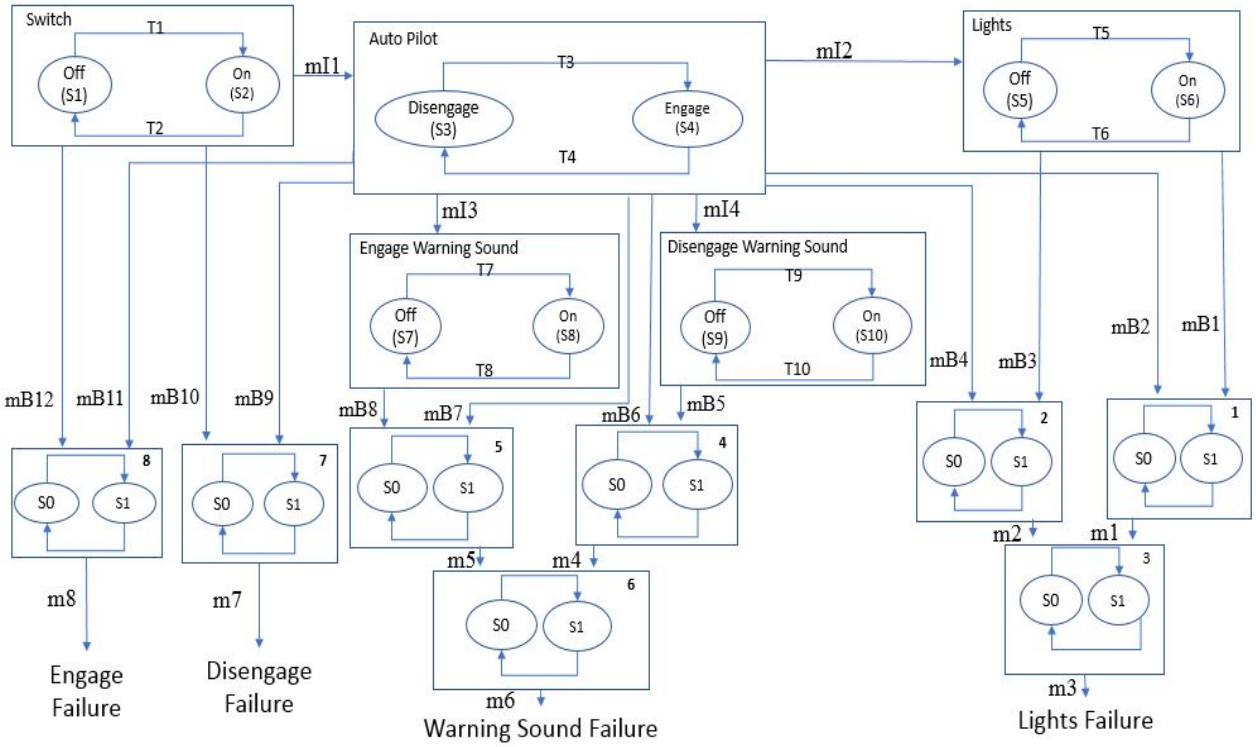

Figure 30. ICEFSM model for an Autopilot Awareness System

Table 5. Failures-Test paths mapping

F.ID	Failure Type	Node in FT	Event ID	Gate ID	Message ID	ICEFSM Test Path
1	Engage Failure	BF- Disengage.BFEventCond	eB11	G8	mB11	10
2	Disengage Failure	BF- Engage.BFEventCond	eB9	G7	mB9	9
3	Warning Sound Failure	BF- Disengage Warning Sound Off.BFEventCond, BF- Engage Warning Sound Off.BFEventCond	eB5,eB7	G4,G5	mB5,mB7	6,5
4	Lights Failure	BF- Lights On.BFEventCond,BF-Lights Off.BFEventCond	eB1,eB3	G1,G2	mB4,mB3	2,1

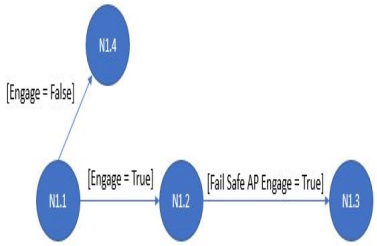

Figure 31. Mitigation Model for Engagement failure

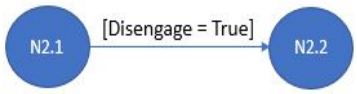

Figure 32. Mitigation Model for Disengagement failure

Figure 33. Mitigation Model for Aural Warning failure

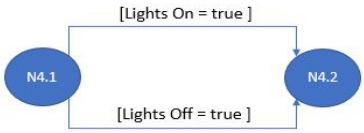

Figure 34. **Mitigation Model for Lights failure**

Generate Safety Mitigation test paths (SMT)

First, we select a failure coverage criteria. Lets assume we chose to cover All tests, All unique nodes and all applicable failure, which is Criteria 2 as illustrated in the Fail-safeMBT description in the background section. Using the applicability matrix in Table 6, and test paths generated from the CEFSM model shown in Table 3, we apply Criteria 2 to cover cover All tests, All unique nodes and all applicable failure. The result is a set of position failure pairs shown in Table 8.

Second, concatenating all test paths as follows: S1 S2 S4 S6 S3 S4 S8 S7 S1 S2 S1 S3 S4 S3 S9 S3 S4 S3 S5 S5 S6 S5 S9 S10 S9 S7 S8 S7, gives us 28 possible positions to inject failures and proper mitigation. Using Table 8, Table 7, Mitigation models and following the weaving rules we procedure illustrated in the safety mitigation test path generation subsection in the Fail-safeMBT description, we construct safety mitigation test suite as illustrated in Table 9.

Table 9. **Safety Mitigation Test Paths**

NO	Position	Failure	SMT	CEFSM Test Path
1	2	1	S1 n1.1 n1.2 n1.3	t1
2	2	1	S1 n1.4 S2 S4 S6	t1
3	3	2	S1 S2 n2.1 n2.2 n2.3 S4 S6	t2
4	14	2	S1 S2 S1 S3 S4 n2.1 n2.2 n3.3 S4 S3 S9	t3
5	8	3	S3 S4 S8 n3.1 n3.2 S7	t2
6	9	3	S3 S4 S8 S7 n3.1 n3.2	t2
7	16	3	S1 S2 S1 S3 S4 S3 S9 n3.1 n3.2	t3
8	24	3	S9 n3.1 n3.2 S10 S9	t6
9	25	3	S9 S10 n3.1 n3.2 S9	t6
10	27	3	S7 n3.1 n3.2 S8 S7	t7
11	28	3	S7 S8 n3.1 n3.2 S7	t7
12	7	4	S3 S4 n4.1 n4.2 n4.3 S8 S7	t2
13	13	4	S1 S2 S1 S3 S4 n4.1 n4.2 n4.3 S3 S9	t3
14	20	4	S3 S4 S3 S5 n4.1 n4.2 n4.3	t4
15	22	4	S5 S6 n4.1 n4.2 n4.3 S5	t5

Deriving Substantiation for Fail-safeMBT Compliance

Since Fail-safeMBT is a model-based methodology, we will be mainly considering certificate DO-331 which provides guidance and requirements for using modeling in the avionic software development process in addition to DO-178C certificate. In DO-331, the verification process consists of analysis, review and testing. Figure 35 adopted from [27] shows the activities mandated by DO-178C to fulfill its objectives (the labels on the arcs). Since Fail-safeMBT, is based on executing test cases generated using test paths against the system under test, we are looking for Fail-safeMBT to contribute to the compliance process and provide traceability between the executable object code to the high-level requirements. In Figure 35, verification of the requirements is shown in the two white boxes with the blue border lines. The boxes with the red borders are the verification that Fail-safeMBT could achieve.

In an effort to bridge the gap, first we establish substantiation for compliance by mapping Fail-safeMBT outputs from its activities to requirements that DO-178C/DO-331 mandates. Table 10 shows what parts of the outputs of Fail-safeMBT could be linked to DO-178C activities and objectives in the verification process. These Fail-safeMBT process outputs are test paths from the CEFSM model, test paths from the ICEFSM model, test paths for safety mitigation tests, and failure-test paths mapping. In DO-331, the testing elements we found related to testing activity requirements are testing the normal behavior, testing for robustness, traceability, and independence. Section 6.4 in DO-331 states that testing should be requirement-based which includes normal and robustness test cases. Section MB 6.1.e in DO-331 states that the executable object code should be robust with respect to the software requirements such that it can respond properly to abnormal inputs and conditions. Since Fail-safeMBT uses system requirements and safety requirements to build the models for testing the system, it will conform with Section 6.4 in DO-331 which emphasizes on requirement-based testing. However, to reach full compliance, several objectives should be accomplished. In the verification of verification results, 12 objectives should be accomplished. Objectives 10-12 are related to the use of simulation. Since Fail-safeMBT does not use simulation, we will eliminate these objectives. Also, objectives 1 and 2 will not be addressed since they are review activities. Seven objectives are left to be checked if Fail-safeMBT can comply with. Fail-safeMBT is a black-box testing technique; therefore, objectives 4,5,6,7,8, and 9 will not be fulfilled since they are source code testing related objectives. Objective 3 is the only candidate left so far for Fail-safeMBT to comply with. In figure 36, we show a conformance argument using GSN to argue about Fail-safeMBT compliance with DO-331 objectives. The argument is formed as follows: Claim 001 is that Fail-safeMBT can meet all objectives in table MB.A-7 in DO-331. The outputsData strategy is that we argue by demonstrating a substantiation from Fail-safeMBT outputs. The only objective we claim that Fail-safeMBT could meet is objective 3 Test Coverage of high-level requirements is achieved. In figure 37, we support this claim by showing that there is at least one test path produced by Fail-safeMBT that can be used to create test cases to test a specific requirement of the SUT. The evidence provided to support the claim are substantiated from Fail-safeMBT outputs: 1) Table 3, test paths from CEFSM model for testing normal behavior, 2) Table 4, test paths from ICEFSM for failure testing, 3) Table 5, failure to test paths mapping to provide traceability and 4) Table 9, the safety mitigation test paths. Regarding the traceability property, Section 6.5 in DO-331 about software verification process traceability

Table 8. All tests, all unique nodes, all applicable failures

F/S	TP_1				TP_2				TP_3							TP_4				TP_5				TP_6			TP_7		
	s_1	s_2	s_4	s_6	s_3	s_4	s_8	s_7	s_1	s_2	s_1	s_3	s_4	s_3	s_9	s_3	s_4	s_3	s_5	s_5	s_6	s_5	s_9	s_{10}	s_9	s_7	s_8	s_7
f_1	1																											
f_2		1											1															
f_3							1	1							1								1	1		1	1	
f_4					1						1								1		1							

mandates that a trace data should exist between test cases and software requirements. Since Fail-safeMBT outputs tables 5 and 9 document what test path is used to cover each failure, traceability will be provided between software requirements and the test cases generated using Fail-safeMBT test paths. Regarding independence, in the verification process, DO-331 mandates that some objectives should be achieved independently depending on the safety level of the SUT. Since Fail-safeMBT is a model-based testing methodology that uses system requirements a different person or organization other than the developers can apply its testing activities if the same requirements are used. Therefore, Fail-safeMBT can provide independence.

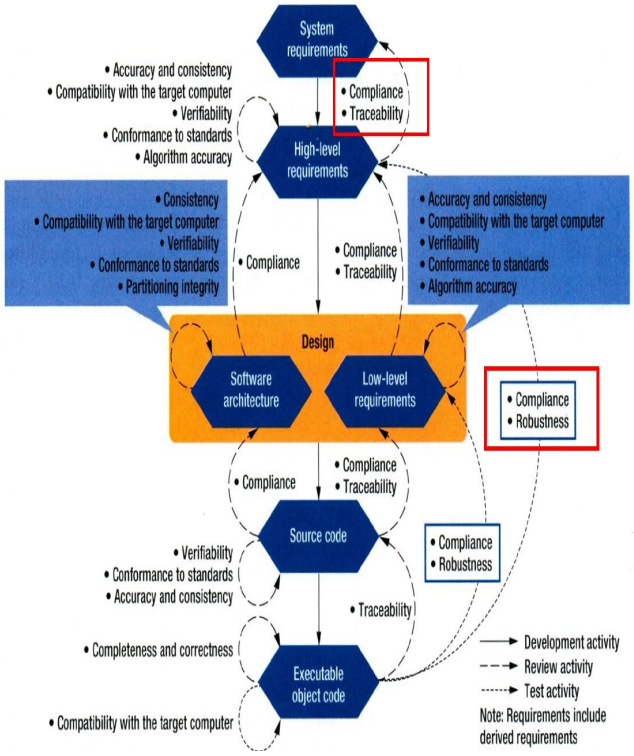

Figure 35. DO-178C activities for fulfilling certification objectives in relation to Fail-safeMBT

Findings

In this research, we were able to answer the research questions as follows: Fail-safeMBT is partially in compliance with DO-178C/DO-331. Fail-safeMBT could be used to provide test paths for test cases generation. These test cases are targeting testing system safety and its mitigation requirements. In avionic standards, testing is required to be conducted in a requirement-based testing manner. As a model-based safety testing technique, Fail-safeMBT uses the system requirements in building the model for testing, therefore, it can provide requirement-based testing with independence. In addition, test cases generated from Fail-safeMBT test paths could be traced to the system requirements under test as all its test paths outputs in the processes are mapped to a specific requirement. However, regarding objectives in table MB.A-7, objective 3 was the only one that Fail-safeMBT met. Objectives 4,5,6,7,8 and 9 are not achievable using Fail-safeMBT due to the fact that Fail-safeMBT is a black-box testing technique. Objectives 1 and 2 are not addressed in this study because they are review activities. As a result, Fail-safeMBT contributes partially to the compliance process. Fail-safeMBT should be further improved by adding white-box testing activities to cover more certification requirements related to source code testing to be recommended as a testing methodology in the avionic domain. In general, any black-box based testing methodology will never fully comply with the standards in the avionic domain.

4. CONCLUSION

In this paper, we presented Fail-safeMBT, a model-based testing methodology for testing safety critical systems as nine-tasks processes using SPEM2.0, then we applied Fail-safeMBT on some of the Autopilot system functionalities. We examined Fail-safeMBT with respect to DO-178C/DO-331 focusing on the compliance of the verification activities. From Fail-safeMBT processes outputs, we have argued about the elements of compliance with the standards related verification activities and objectives by using GSN. Fail-safeMBT showed a partial compliance with the standards, specifically as a requirements-based testing technique, providing complete coverage in testing high-level requirements in addition to providing the independence property when required by the safety level. However, Fail-safeMBT still lacks high appropriateness in the avionic domain regarding compliance with source code testing activities and objectives as a result of Fail-safeMBT being a black-box testing technique. Therefore, Fail-safeMBT should be further improved by adding more testing tasks or combining it with other testing techniques to achieve full certification compliance in the avionic domain.

Table 10. Fail-safeMBT outputs mapping to DO-178C/DO-331 elements

Fail-safeMBT Table	Purpose	Step in SPEM	Related DO-178C/DO-331 parts	Explanation
Table 3	Test the normal behavior of the system	Test Paths Generation and Failure Mapping	Section 6.4.2, Section 6.4.3, Section MB 6.1.e, Table MB.A-7	Requirements based Testing, Normal and Robust Testing
Table 4	Test for failures (Robustness)of the system	Test Paths Generation and Failure Mapping	Section 6.4.2, Section 6.4.3, Section MB 6.1.e, Table MB.A-7	Requirements based Testing Normal and Robust Testing
Table 5	To trace any failed test case to the specific failure	Test Paths Generation and Failure Mapping	Section 6.5	Test cases to Software requirements traceability
Table 9	To test for system proper mitigation behavior when a failure occurs	Safety Mitigation Test Paths Generation	Section 6.4.2, Section MB 6.1.e, Table MB.A-7	Requirements based Testing

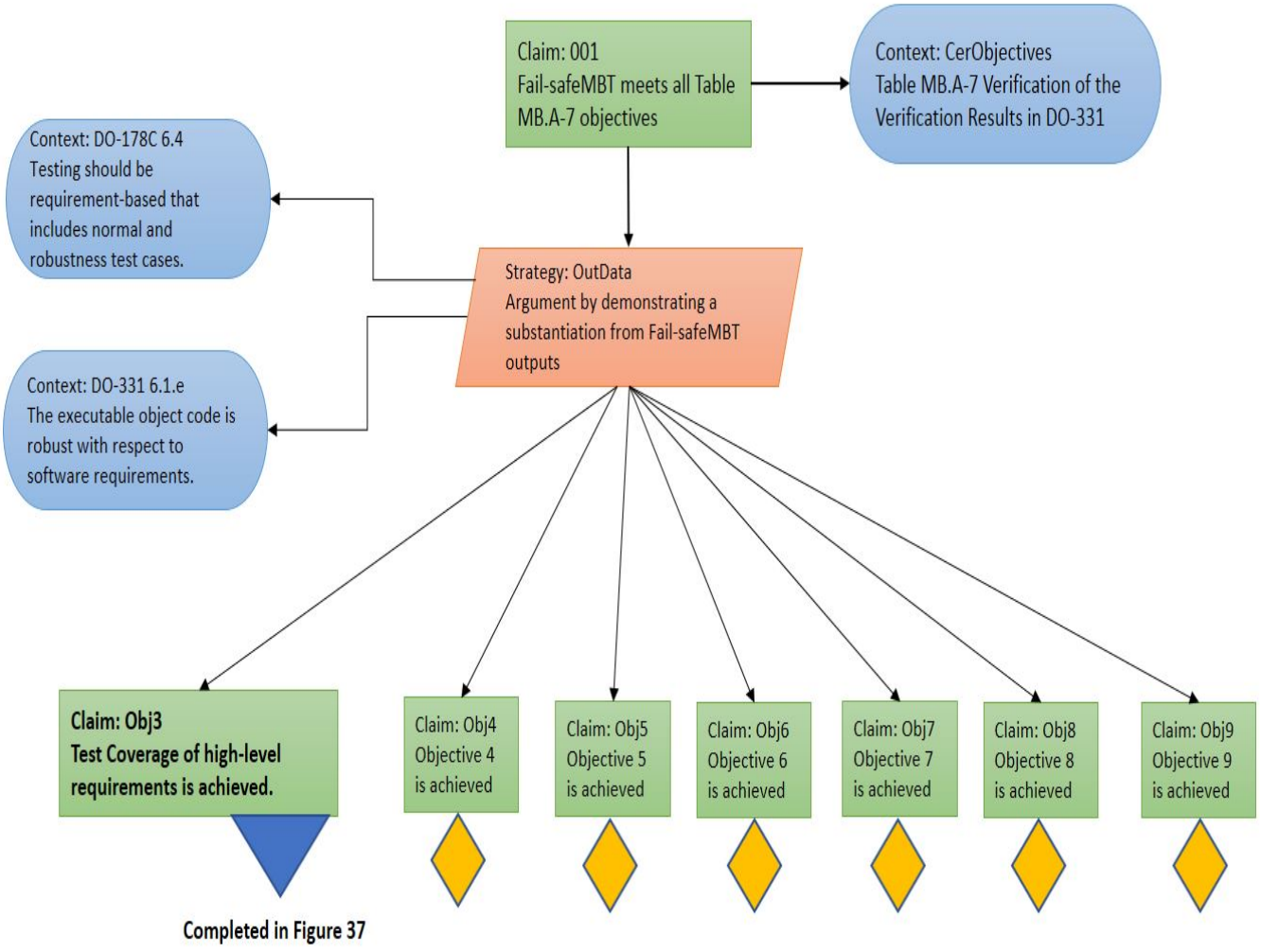

Figure 36. Conformance argument shows how Fail-safeMBT meets one of DO-331 objectives.

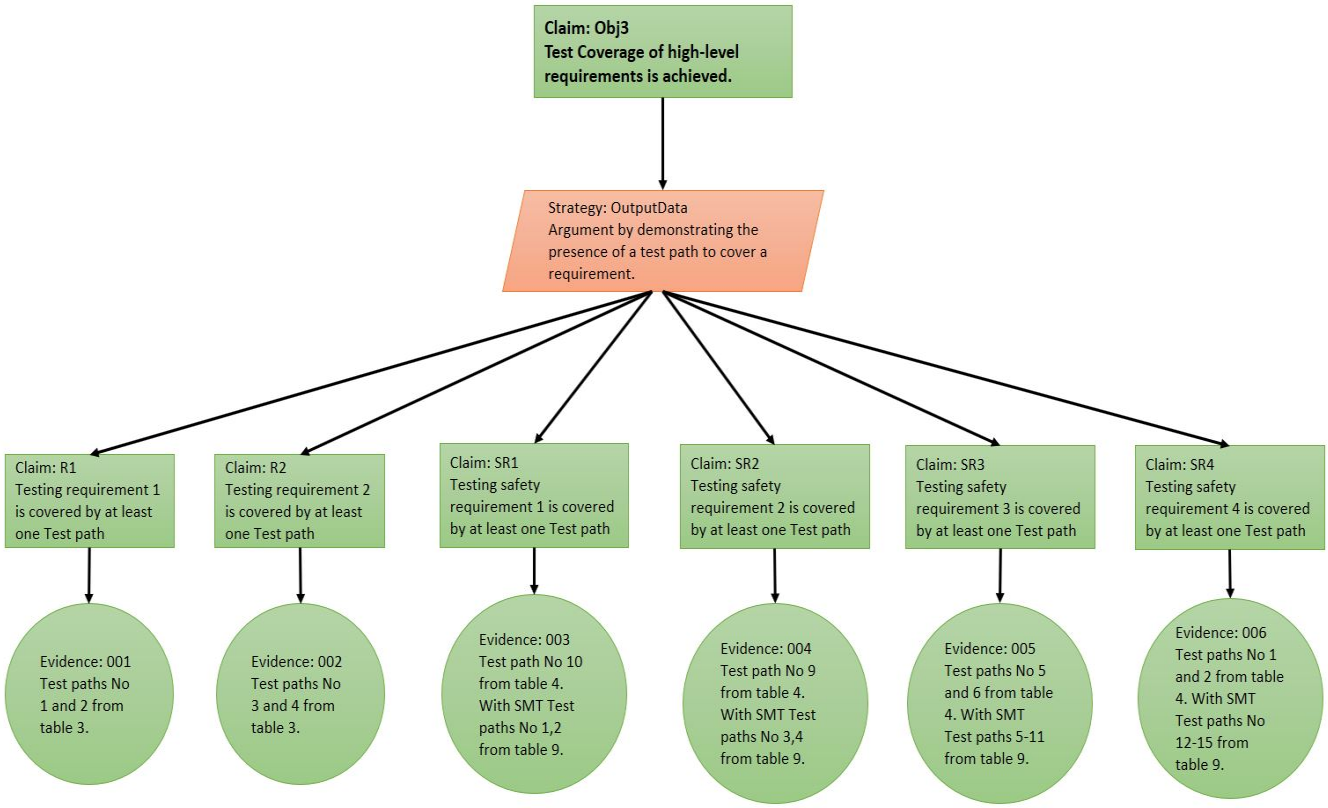

Figure 37. Conformance argument shows how Fail-safeMBT meets objective 3 of Table MB.A-7 in DO-331.

ACKNOWLEDGMENTS

This work was partially supported in part by NSF grant #1439693 to the University of Denver. The author B. Gallina is financially supported by the EU and VINNOVA via the ECSEL JU project AMASS (No. 692474).

REFERENCES

[1] U. Zdun, A. Bener and E. L. Olalia-Carin, "Guest Editors' Introduction: Software Engineering for Compliance," In IEEE Software, vol. 29, no. 3, pp. 24-27, May-June 2012.

[2] P. Graydon, I. Habli, R. Hawkins, T. Kelly and J. Knight, "Arguing Conformance. In IEEE Software," vol. 29, no. 3, pp. 50-57, May-June 2012.

[3] C. Holloway, "Making the Implicit Explicit: Towards an Assurance Case for DO-178C," Proc ISSC, Boston, MA, ISSS, 2013.

[4] RTCA DO-178C (EUROCAE ED-12C), Software Considerations in Airborne Systems and Equipment Certification, RTCA Inc, Washington DC, 2013.

[5] RTCA DO-331, Model-Based Development and Verification Supplement to DO-178C and DO-278A, RTCA Inc, Washington DC, 2011.

[6] L. Rierson, Developing safety-critical software: a practical guide for aviation software and DO-178C compliance. CRC Press, First edition, 2013.

[7] A. Avizienis, J. Laprie, B. Randell, C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," IEEE transactions on dependable and secure computing 1.1, pp. 11-33, 2004.

[8] B. Lerner, S. Christov, L. Osterweil, R. Bendraou, U. Kannengiesser and A. Wise, "Exception Handling Patterns for Process Modeling," In IEEE Transactions on Software Engineering, vol. 36, no. 2, pp. 162-183, March-April 2010.

[9] A. Gario, A. Andrews and S. Hagerman, "Testing of safety-critical systems: An aerospace launch application," In IEEE Aerospace Conference, pp. 1-17, 2014.

[10] A. Anneliese, S. Elakeili, A. Gario, S. Hagerman, "Testing proper mitigation in safety-critical systems: An aerospace launch application," In IEEE Aerospace Conference, pp. 1-19, 2015.

[11] D. Dewi, "Are we there yet? A Practitioners View of DO-178C/ED-12C," Advances in Systems Safety. Springer London, pp. 293-313, 2011.

[12] OMG, Software & systems Process Engineering Metamodel (SPEM), v 2.0. Full Specification formal/08-04-01, Object Management Group, 2008.

[13] G. Baumgarten, M. Rosinger, A. Todino and R. de Juan Marn, "SPEM 2.0 as process baseline Meta-Model for the development and optimization of complex embedded systems," 2015 IEEE International Symposium on Systems Engineering (ISSE), pp. 155-162, 2015.

[14] GSN: Community Standard Version 1, 2011.

[15] T. Kelly and R. Weaver," The goal structuring notation - A safety argument notation," In the proceedings of the dependable systems and networks workshop on assurance cases, Citeseer, 2004.

[16] B. Gallina and A. Andrews, “Deriving verification-related means of compliance for a model-based testing process,” Digital Avionics Systems Conference (DASC), IEEE/AIAA 35th, 2016.

[17] B. Gallina, “A model-driven safety certification method for process compliance,” Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium, 2014.

[18] www.faa.gov

[19] C. Ericson, Hazard Analysis Techniques for System Safety, John Wiley & sons Inc., 2005.

[20] M. Sánchez and M. Felder, “A Systematic Approach to Generate Test Cases based on Faults,” In Argentine Symposium in Software Engineering, Buenos Aires, Argentina, 2003.

[21] R. Nazier and T. Bauer, “Automated Risk-Based Testing by Integrating Safety Analysis Information into System Behavior Models,” In IEEE 23rd International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 213-218, 2012.

[22] C. Cârlan ,B. Gallina , S. Kacianka and R. Breu, “Arguing on Software-Level Verification Techniques Appropriateness,” In International conference on Computer Safety, Reliability, and Security. SAFECOMP 2017. Lecture Notes in Computer Science, vol 10488, pp. 39-54, Springer, Cham, 2017.

[23] R. Bloomfield and P. Bishop, “Safety and Assurance Cases: Past, Present and Possible Future - an Adelard Perspective,” Making Systems Safer, Springer, London, pp 51-67, 2010.

[24] A. Tribble and S. Miller, “Software intensive systems safety analysis,” In the IEEE Aerospace and Electronic Systems Magazine, vol. 19, no. 10, pp. 21-26, Oct. 2004.

[25] E. Leaphart,B. Czerny,J. D’Ambrosio, C. Denlinger, and D. Littlejohn, “Survey of Software Failsafe Techniques for Safety-Critical Automotive Applications,” SAE World Congress, 2005.

[26] P. Ammann and J. Offutt. Introduction To Software Testing. Cambridge University Press, New York, USA, First edition, 2008.

[27] B. Monate, E. Ledinot, H. Delseny, V. Wiels and Y. Moy, “Testing or Formal Verification: DO-178C Alternatives and Industrial Experience,” IEEE Software vol. 30, p. 50-57, 2013.

BIOGRAPHY

Aiman Gannous *received his B.S. degree in 2001 and M.S. degree in 2008 in Computer Science from Garyounis University, Benghazi, Libya. He is currently working toward his Ph.D. degree in computer science at the University of Denver. His current research activities include integration and combinatorial testing of robotics and autonomous systems.*

Anneliese Andrews *Dr. Anneliese Andrews is Professor of Computer Science at the University of Denver. Before joining the University of Denver, she held the Huie Rogers Endowed Chair in Software Engineering and served as Associate Director of the School of Electrical Engineering and Computer Science at Washington State University. Dr. Andrews is the author of a text book and over 230 articles in the area of Software and Systems Engineering, particularly software testing, system quality and reliability. Dr. Andrews holds an MS and PhD from Duke University and a Dipl.-Inf. from the Technical University of Karlsruhe. She served as Editor in Chief of the IEEE Transactions on Software Engineering. She has also served on several other editorial boards including the IEEE Transactions on Reliability, the Empirical Software Engineering Journal, the Software Quality Journal, the Journal of Information Science and Technology, and the Journal of Software Maintenance. Dr. Andrews is the DU site Director of a NSF Industry/University Collaborative Research Center: Robots and Sensors for the Human Wellbeing.*

Barbara Gallina *Dr. Gallina is Associate Professor of Dependable Software Engineering at Mälardalen University, where she leads the Certifiable Evidences and Justification Engineering group. She has been visiting researcher at Scania AB. She has been member of several program committees related to dependability such as SafeComp, ISSRE, EDCC, COMPSAC, QUORS, WoSoCER, SASSUR, ReSACI, ISSA. Dr. Gallina is the author of over 50 articles in the area of Dependable Software Engineering.*