Proceedings of the AIAA Scitech Forum 2019, San Diego, 7-11 January 2019.
DOI: 10.2514/6.2019-0553

Incorporating Safety in Early (Airframe) Systems Design and Assessment

Sergio Jimeno Altelarrea,[*] Arturo Molina-Cristóbal,[†] Atif Riaz[‡] and Marin D. Guenov[§]
School of Aerospace, Transport, and Manufacturing
Cranfield University, Cranfield, Bedfordshire, MK43 0AL, United Kingdom

Presented is a novel framework for incorporating safety analysis in early systems architecture design. Traditionally, a systems architecture is first defined by the architects and then passed to safety experts, who manually create artefacts such as Function Hazard Analysis (FHA) or Fault Tree Analysis (FTA) for safety assessment. The problem with this manual approach is that if the architect modifies the systems architecture, then the whole safety assessment process needs to be repeated, which is tedious and time consuming. To overcome this limitation, the proposed framework automates the creation of safety models such as FHA and FTA by utilizing the Requirement, Functional, Logical, and Physical (RFLP) systems engineering paradigm. The framework supports three main activities. First, the safety targets are determined by performing a FHA of the architecture and the Requirements view is updated. Second, compliance with the safety requirements is analyzed using dynamic fault trees, automatically generated from the Logical view. Interactive visualization techniques are proposed to interpret the safety results, e.g. highlighting the greatest contributors to the probability of failure. Third, an algorithm is developed that enables the designer to interactively improve the architecture's safety by introducing more reliable components or increasing redundancy. The concept is illustrated with a representative example, where the environmental control system of a civil aircraft is studied from a safety point of view.

I. Introduction

System safety is of paramount importance in all industries, and particularly in aerospace. Specifically, in relation to (civil) aircraft, there are many failure conditions which may lead to a potentially fatal accident, involving multiple casualties, damage to the aircraft, the surrounding area of the accident and which incur substantial monetary losses.

Currently, safety analyses are performed manually, based on informal models and various documents regulating the certification of aircraft, such as the Certification Specifications CS-25 [1], which establish safety objectives on the different parts of the aircraft in order to keep the probability of catastrophic accidents to a minimum. However, this manual approach brings unnecessary subjectivity and lack of consistency to the analyses [2]. Furthermore, safety assessment usually takes place later on in the product development process, after the design is finalized [3]. This, in turn may lead to costly redesign efforts.

Different approaches to this problem have been proposed in recent years focusing on automating safety analyses such as the fault tree analysis (FTA). Although they have many aspects in common, differences between the methods can be found mainly in two areas: 1) the input to the algorithms, the underlying data structures representing the architecture e.g. SysML diagrams or AADL models; and 2) its level of the detail, which appears to be correlated with the design phase. With respect to the latter, the methods can be abstracted in two categories: those who use only the connections between components, and those who consider additional failure information such as different component states, events and transitions from one state to another.

In the first category, Mhenni et al. [3] propose the integration of safety analysis within a systems engineering approach by using SysML internal block diagrams for automatic component-based fault tree generation. Roth et al.

[*] Research Student, Center for Aeronautics, Cranfield MK43 0AL, United Kingdom.
[†] Lecturer, Centre for Aeronautics, Cranfield, MK43 0AL, United Kingdom.
[‡] Research Fellow, Centre for Aeronautics, Cranfield, MK43 0AL, United Kingdom.
[§] Professor, Head of the Centre for Aeronautics, Cranfield, MK43 0AL, United Kingdom, AIAA Senior Member.

[4] found their method for automatic FTA on the Structural Complexity Management methodology[**], producing fault trees based on the system functions. One addition limitation of the later is that redundancy needs to be included in the FTA manually.

Amongst the method that use additional failure information, Xiang et al. [5] enable automatic fault tree synthesis and reliability analysis by combining internal block diagrams and sequence diagrams with a reliability configuration model and a static fault tree model. Papadopoulos et al. [6] extend the Safety Argument Manager[††] to simplify the development of FTAs for complex programmable and electronic systems. Delange and Feiler [7] introduce an Error-Model annex to the AADL language to model failure states of components, failure propagation and internal and external failure events, thus enabling the automatic computation of different safety analyses. FTA capabilities similar to the previous approach are also provided by Li and Li [8], who base their safety analysis tools on Altarica, and by Majdara and Wakabayashi [9], who facilitate automatic fault tree synthesis by developing their own modelling language. Tajarrod and Latif-Shabgahi [10] propose an extension of Simulink models containing functional (failure) and behavioral information, in order to automatize FTA. Simulink is also the preferred approach for Papadopoulos and Maruhn [11] where fault propagations are modelled for each component. Without explicitly obtaining fault trees, Schallert [12] proposes a method to obtain FTA results from extended Modelica models.

One of the main limitations, common to all the methods, is that they consider safety analysis as a one-way only process. Given a particular design, the methods can be used to evaluate its safety characteristics and compliance with a pre-existing set of requirements. However, the learnings from this process are not used to improve the architecture, with the exception of [6] which considers the modification of the design but does not propose any methodology. We believe that safety analysis should be viewed as a two-way process which is repeated through the design process and improves the safety characteristics of the design iteration after iteration. In addition, potential modifications made to improve safety, will have implications on the sub-system and system level performances. These implications have been neglected in the all previous methods limiting the applicability of their results. Another shared downside to the methods is that they only consider standard fault trees, as opposed to dynamic fault trees or other more advanced extensions.

Within this context, the aim of the work presented in this paper is to improve safety assessment in the early product development process in order to overcome the previously discussed limitations of existing methods. The scope of the presented work is limited to early design, under the assumption that the requirements, functions, and logical views of the (systems) architecture are defined to a basic level. The physical view is not considered at this level of detail. Computational models for sizing and performance are also out of the scope of this work. The safety analyses are limited to simple operational scenarios, where only one configuration of the system is considered and details about the different phases of the mission are excluded.

The rest of the paper is organized as follows. Section II provides background information and terminology used in the manuscript. The proposed methods for aiding the automation of safety analysis are described in Section III. Section IV demonstrates the application of the proposed approach to a representative test case. Finally, summary, conclusions and plans for future work are drawn in Section V.

II. Background

This section explains the terminology and gives a very brief overview of some of the safety analysis methods referred to in this paper.

A. FHA - Functional Hazzard Assessment [1].

Comprehensive and systematic study of functions to identify failure conditions and classify them according to their severity. It can be done at aircraft or system level, depending of the examined functions. The system levels FHA are used as an input to the system PSSA.

B. PSSA - Preliminary System Safety Assessment [1].

Systematic examination of a system, to determine how failures can cause the FHA functional hazards and how the safety requirements can be met. It employs Fault Tree Analyses or similar tools to determine the latter.

C. SSA - System Safety Assessment [1].

[**] A methodology for handling complex system and structural dependencies by combining Design Structure Matrices and Domain Mapping Matrices [2]

[††] A tool for safety management described in [3]

It is a systematic, comprehensive evaluation of a system, its architecture and installation to show that requirements from the FHA and system PSSA are met. Similar tools to those of the PSSA are employed.

D. FTA – Fault Tree Analysis

Deductive failure analysis that focuses on one undesired event, the top event, and determined its causes in terms of basic events and their relations expressed through logical gates. According to the review paper by Ruijters and Stoelinga [14], two main kinds of fault trees exists, standard and dynamic, as well as several less known extensions. The principal distinction is that standard fault trees can only model system failure through combinations of component failures, whereas dynamic fault trees introduce the notion of temporal sequence of failures.

Standard fault trees only use mainly three types of gates:

- AND: the output event needs all input events to happen (Some variations may include the INHIBIT gate, whose function is equivalent to the AND gate).
- OR: any of the input events occurring is sufficient to make the output event happen.
- K out of N: any sets with k of the N input events is sufficient to make the output event happen.

. Dynamic fault trees add three new kinds of gates:

- PAND: priority AND, specifying the order of input events that leads to the output event.
- FDEP: the first input of the gate acts as a trigger, making all other inputs to occur when it fails.
- SPARE: when the primary unit fails, it is substitute by any of the available spares, which become active.

Regarding the probability of failure of the basic events, standard fault trees use generally inverse exponential distributions [14] $P_f(t) = 1 - \exp(-\lambda t)$, which are determined by the failure rate λ. Dynamic fault trees include a dormancy factor $\alpha \in [0, 1]$ that reduces the probability of failure for inactive components.

E. Importance measures

Importance measure indicate how important is each basic event with respect to the probability of failure of the top event. There exist many importance factors, Dutuit and Rauzy [15] analyze six of them, their mathematical formulation and possible physical interpretation. Here we will focus on two well-known measures.

The Fussell-Vesley importance factor, also known as diagnostic importance factor measures the fraction of the system unavailability that involves the basic event happening.

$$FV \coloneqq \frac{P_f(top \;/\; A = \; 1)}{P_f(top)} \tag{1}$$

where $P_f(top/A = 1)$ is the probability that the top event occurs give that event A do occur ($\mathrm{P_f}(A) = 1$). The Birnbaum importance factor, also called marginal importance factor, indicates the conditional probability that fixing the problem with the event A stops the top event from happening

$$Birnbaum \coloneqq P_f(top \;/\; A = \; 1) - P_f(top \;/\; A = \; 0) \tag{2}$$

where $P_f(top/A = 0)$ is the probability that the top event occurs give that event A does not occur ($P_f(\mathrm{A}) = 0$).

F. RFLP Paradigm, augmented with Computational Domain

It assumes that functional reasoning as part of the systems architecting process is distributed over four notional domains: Requirements, Functional, Logical, and Physical [16]. It is based on the German guideline VDI 2206, "Design methodology for mechatronic systems" [17]. RFLP is augmented with a Computational Domain to provide the capability of automated systems sizing, as proposed by Bile et al. [18]. Additionally, traceability between different view of the architecture is incorporated by Guenov et al. [16], enabling a more effective and interactive design process.

Requirements View: The requirements view displays all the architecture requirements, which represent the stakeholder needs. Requirements can be of functional or performance type, and it is possible to decompose them hierarchically. Requirements are mapped to the functions of the system in the Functional View.

Functional View: This view contains all the architecture functions, actions that the system must perform to meet the stakeholders' needs. Functions are linked to components, and vice-versa. Function support decomposition.

Logical View: The logical view consists of components, solutions satisfying the functions, and their interconnections via ports.

Computational View: Strictly speaking, this is not part of RFLP. It is a notional domain introduced in [18] with its primary purpose to automatically orchestrate the computational code (e.g. steady state models) associated with each component in the logical view into sizing (computational) workflows. Indirectly this domain is used to manage the

dependencies between parameters, i.e., the numerical values that describe some (behavioral) characteristics of the components, and which can be linked to performance requirements.

G. Redundancy

The concept of redundancy was introduced as a means of constructing reliable systems out of unreliable components [19]. Two main kinds of redundant structures have been developed, static redundancy where all the redundant items are on-line contributing to the output, and dynamic structures where only one part of them are on-line and the others wait to be activated in case another component fails, usually in a dormant state [20]. Analogous strategies have been also developed for improving the reliability of software, e.g. N-version programming and recovery blocks [21].

Safety and reliability concerns are of maximum importance for the aerospace industry, and application of redundancy can be found in most of the subsystems. Some examples (taken form Moir and Seabridge [22]) are the flight control system, with redundant actuators and in case of fly by wire redundant computers and buses, the hydraulic system with various hydraulic channels, or the power distribution system, with redundant distribution lines and various sources including emergency ones.

III. Proposed Approach

In order to improve safety assessment in the early product design, a novel safety framework, based on the RFLP paradigm, is proposed. As shown in Fig. 1, safety is regarded in the framework as a two-way process. It uses the architecture definition as an input to safety analyses such as FTA or FHA. In turn, their results are used to update the architecture, for example by adding new requirements or logical components to keep safety compliant. These changes may have an impact on the system performance which might demand even further modification, which in turn affect safety and so forth. As discussed in [18], the RLFP paradigm augmented with a computational view provides a means of assessing such impact. Nevertheless, the main focus of this paper is on the safety part.

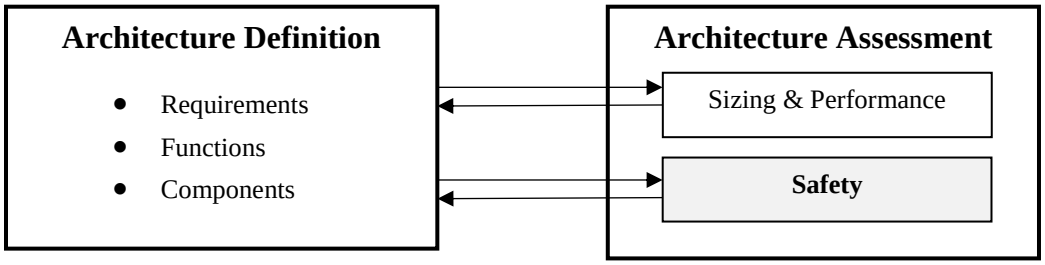

Fig. 1 Overview of the Architecting process including safety

In this section the main parts of the proposed framework are introduced. Fig. 2 shows an overview of the framework (in light blue) and its links with existing RFLP elements. The main purpose of these extensions is to support the automation of FTAs and partial automation of FHAs from early design stages, e.g. in the scope of PSSAs or SSAs. This framework not only enables safety assessment, but, in combination with the existing traceability between RFLP domains, it also helps keeping it up to date as the design evolves. Additionally, it can provide guidance to the architect in the process of modifying the architecture to meet the safety requirements, as indicated in the link between FTA and L in Fig. 2.

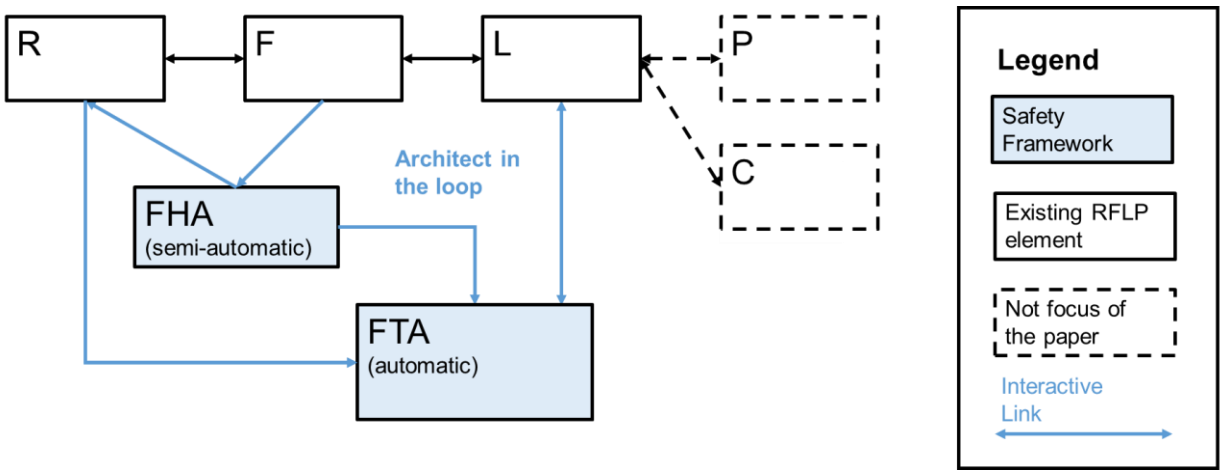

Fig. 2 Overview of the RFLP Safety Framework

A. Functional Hazzard Assessment

The first part of the framework aims to support the creation of FHAs. One of the first steps of a FHA is obtaining a list of functions of the system [23], to analyze the hazards related to them. This list is compiled automatically by traversing part of the Functional view of the architecture in its hierarchical form. In this form, functions are organized using parent child relationships (with at most one parent), resulting in a data structure known as tree[‡‡]. The scope of the FHA is determined by the selected components to analyze, e.g. the components forming the subsystem being studied. The functions that map to the selected components, plus all their children are included in the FHA.

It is recommended to use a worksheet to perform this kind of analysis [23], consequently this is the approach followed here. The views of the architecture contain enough information to fill some of the columns of the worksheet such as the function and the hazard number. The rest of the FHA fields need to be filled in manually. Even though this prevents full automation of the assessment, the information in the functional view can provide valuable support, for example by helping to obtaining the potential functions affected by the failure of another one, as suggested by Wilkinson and Kelly [24]. From the point of view of enabling Fault Tree Analysis only one additional field needs to be completed by the architect, "Classification", as it determines the level of detail necessary in further analysis (e.g. FTA) and the required probability target.

For each requirement mapped to a function considered in the FHA, a child safety requirement can be created and appended to it. The safety requirements will display a message describing qualitatively or quantitatively the severity of the requirement, following to the definitions specified in CS-25 [1], Book 2 AMC 25.1309. For example, for a failure condition classified as Major, the qualitative requirement message will be "No more frequent than remote" and the quantitative one will display "With a maximum average probability per flight hour of order 10^{-5} or less". These safety requirements store the probability target that will be used in future analysis like FTA. All mapped functions plus their children are considered for one requirement, in case their number is greater than, only strictest of them is displayed.

In this way, the FHA is used to update the requirements view in order to reflect safety concerns. Furthermore, these requirements are automatically linked to the elements in other views, enable the automatic setup of further analysis such as Fault Tree or Dependency Diagram analyses.

B. Fault Tree Analysis

During a system PSSA the architecture needs to be evaluated to show, among others, how item failures lead to the various failure conditions, and how the qualitative and quantitative objectives for those conditions. Fault Tree Analysis is the first of the three recommended tools by the ARP4791 [13] in order to perform the previous tasks. Similar activities need to be performed during an SSA, although it is generally more detailed as it occurs later in the design and incorporates the results from the PSSA.

[‡‡] A rooted tree [4] is a connected, acyclic, undirected graph in which one of the vertices the highest-level function of the architecture is distinguished from the others, creating a hierarchy via parent/child relations. Multifunctional architectures are represented by a set of rooted trees, also called a forest.

The logical view can be used to generate fault trees automatically by using an algorithm which is described later in this section. This fault tree, will be based on the system components, whose faults correspond to the basic events in the tree; and connections between components, which model the different kinds of redundancy present in the system, and translate into one of the various types of logical gates described in the previous section.

Some additional elements, not considered part of the RFLP paradigm so far [16] are necessary to enable some FTA features. For quantitative analysis, the definition of the logical components in the architecture must be extended with a component probability of failure and a dormancy factor in case that dynamic redundancy is employed. To indicate advanced cases of redundancy, it is necessary to extend the connections with information about the redundancy type. These are the only two required changes to enable the level of FTA described in this paper.

Algorithm for tree creation

The algorithm for creating the fault tree from the logical view (Table 1), traverses the tree in as similar way to the one proposed by Mhenni et al. [3], but the rest of the elements are different. First, the algorithm presented here is designed to handle RFLP logical views instead of the SysML internal block diagrams. Second, it can handle more variety of redundancy cases, such as dynamic redundancy. Finally, unlike in the Mhenni et Al. algorithm, the construction of the tree occurs in a bottom up way. The subtrees corresponding to the children of a gate are built before the gate itself. In this way it is possible to omit single output gates, which are more compact and therefore easier to interpret.

The algorithm CREATE-TREE is used to initialize the tree and the auxiliary sets of parent components in the recursion tree and visited components, and to call the algorithm CREATE-TREE-RECURSIVE-COMPONENT. This second algorithm needs the following inputs: an *output* of any of the components in the logical view, which will provide the name for the output of the gate to be added to the tree and the component to be modelled by this gate; the *tree* itself, to which the gate will be added; the *logical view*, which will provide all the necessary information about the architecture components and interconnections; the set of parent component in the recursion tree, to avoid introducing cycles in the tree; and the set of visited components to avoid analyzing components more than once. First the algorithm gathers the necessary information to construct the gate that describe a failure in the output. This is assumed to happen when the component that provides the output of any of its inputs fail, hence the type being set to OR. This information is provided respectively by GET-COMPONENT and GET-INPUTS. Before constructing the gate, the subtrees rooted on the gates inputs are determined by calling CREATE-TREE-RECURSIVE-CONNECTION for each one of them. Finally, these subtrees and the basic event of component fail are put together with the OR gate.

The algorithm, CREATE-TREE-RECURSIVE-CONNECTION works analogously to the previous one, but instead of receiving a model output receives a model input. This input is used to get the connection to be modelled by the gate using GET-CONNECTION. The type of the gate is determined by redundancy type specified in the connection, e.g. AND when any output can be used, k/N for voting systems where k correct inputs out of N are needed, and SPARE gates for modelling dynamic redundancy. Then, the outputs reached are obtained by get-outputs, the outputs will be filtered according to whether they belong to the set of parents and the logic determined by the type of gate. E.g. for and OR gate, removing the gate inputs contained in the set of parents is enough, but in case of an AND gate if any of the inputs is false, all of them are remove since the gate output is always false. Before constructing the gate, the subtrees rooted on the gates inputs are determined by calling CREATE-TREE-RECURSIVE-COMPONENT for each one of them. Finally, these subtrees describing the failure of the component inputs are put together with the gate.

It is important to note that the neither of the two previous algorithms is recursive by itself as both can either call the other or just return without calling any in the base case (component with no inputs or connection that does not reach any output). However, if both of them are considered together, we can see that the first one calls the second, which I turn calls the first, behaving effectively in a recursive manner.

The update of the fault tree happens in the ADD-GATE algorithm. If there is more than one input, the algorithm will create a gate of the selected type is created, producing the desired output, and depending on the selected inputs. It will append the gate to the tree, updating set of events E and the set of gates G and it will return the output, which represent the gate. Otherwise, no gate is created and the only input (or null set) is returned to be appended to a gate higher in the fault tree.

The approach used here for creation of the fault tree shares common subtrees, using only one instance in the underlying data structure (acyclic graph). This approach is the one used by Ruijters and Stoelinga [14] to model fault trees as it allows for a more compact directed acyclic graph, instead of mimicking its graphical representation which usually models common subtrees either by duplication or by separating the subtree and referencing in through the use of transfer in/out gates. The conventional tree layout can be recovered when rendering for visualization.

Table 1 Algorithm for FTA creation

```
CREATE-TREE (logical-view, top-event)
1   tree = ⟨∅, ∅⟩ // a fault tree is described by the tuple T = ⟨E, G⟩, where E is the set of events (basic and intermediate) and G is the set of gates
2   parents = ∅ // set of parent nodes in the tree, will help to avoid loops in the tree
3   visited = ∅ // set of visited nodes in the tree, will help to avoid visiting a node more than once
4   return CREATE-TREE-RECURSIVE-OUTPUT (logical-view, top-event, tree)

CREATE-TREE-RECURSIVE- COMPONENT (output, tree, logical-view, parents, visited)
1   component = GET-COMPONENT (logical-view, output)
2   parents = parents ∪ component // the component will be a parent in the subsequent recursive calls
3   inputs = GET-INPUTS (logical-view, output )
4   gate-inputs = ∅
5   for each input ∈ inputs
6       gi = CREATE-TREE-RECURSIVE-INPUT (input, fault, logical-view, output, tree, parents, visited)
7       gate-inputs = gate-inputs ∪ gi
8   parents = parents − solution // exclude from the set, the component is not a parent any longer
9   visited = visited ∪ component // the component is marked as visited so it is not visited again
10  return ADD-GATE (tree, output, component ∪ gate-inputs , OR) // Failure of the component or any of its inputs

CREATE-TREE-RECURSIVE-CONNECTION (input, fault-tree, logical-view, output, tree, parents, visited)
1   connection = GET-CONNECTION (logical-view, input)
2   type = GET-GATE-TYPE (logical-view, input)
3   outputs = GET-OUTPUTS (logical-view, input, parents, type) // will exclude connections to parent components
4   gate-inputs = ∅
5   for each output ∈ outputs
6       gi = CREATE-TREE-RECURSIVE- OUTPUT ( output, fault-tree, logical-view, output, tree, parents, visited)
7       gate-inputs = gate-inputs ∪ gi
8   return ADD-GATE (tree, input, gate-inputs , type) // Failure of any of the valid outputs related to the connection

ADD-GATE (tree, output, inputs, type)
1   if |inputs| > 1 // Only add gates with more than one input
2       Create a new gate with name gate
3       gate.type = type
4       gate.inputs = inputs
5       gate.output = output
6       ⟨E, G⟩ = tree
7       E = E ∪ output ∪ inputs
8       G = G ∪ gate
9       Tree = ⟨BE, G⟩
10      return output
11  else
12      return inputs
```

Algorithms for tree evaluation

The algorithm for Fault Tree qualitative evaluation is based of MOCUS algorithm [26]. The main modification is adapting it to our own data structures, a hash table. Once all the minimum cut sets have been obtained, the table allows to retrieve in constant time the minimal cut sets for a desired basic event.

The algorithm for quantitative evaluation is based on the implementation of equations 11.1 in Kececioglu [27]. It is used to obtain the probability of failure of the top event as well as to calculate the ranking of contributors to the fault condition according to the selected importance measures. The higher order terms in the computation of the probability can be truncated according to a selected level of accuracy to be achieved, often called relative cutoff [28], thus reducing the computational time.

The authors acknowledge that there are other algorithms, e.g. using Binary Decision Diagrams, that can provide better performance in some situations. However, their implementation demands more resources and the selected ones are enough to demonstrate the proposed approach to safety evaluation.

C. Interactivity enablers

In order to enable a highly interactive architecting process, the following links between the safety framework and existing RFLP elements are proposed, in a similar fashion to those proposed by Guenov et Al. [16]. The results of the analyses, FHA tables, FTA diagrams and their quantitative and qualitative results are linked to the views of the architecture providing seamless navigation. For example, the FHA table entries navigate to the corresponding function, fault tree basic event map to the corresponding component, gates map to their inputs and, similarly, minimum cut sets map to their elements.

D. Architecture modification for safety purposes

In some occasions, the initial architecture may be found to be not safe enough and thus it needs to be modified to comply with the requirements. This modification might involve substituting components for more reliable alternatives, for which functional-logical knowledge capture mechanism described in [16] can provide support. Another option is to add redundancy to parts of the design, for this purpose, an algorithm to support architects and simplify the process is described next.

Algorithm for redundancy addition

It allows the architect to create redundancy in an interactive manner. The algorithm reduces the effort of applying redundancy to the architecture and prevents possible inconsistencies resulting from a manual duplication process. It automatically handles duplication of components, linking between newly created components and between those and existing ones including the setup of redundancy properties, and mappings between components and other architectural elements such as functions.

In order to apply redundancy, the algorithm requires the user to provide information to solve any issue that might appear (e.g. if an already redundant connection is found it can be further replicated or combined with the one to be created) and to limit the number of components included. This way all the necessary decisions are made explicit and considered before actually modifying the architecture. This not only increases the architect's awareness of the changes to be made but also eliminates the possibility of introducing some modifications that are found later to be wrong due to a not considered issue.

The process starts by selecting the component to be made redundant, along with the number of channels and type of redundancy to be created. Then the framework will traverse the architecture including or excluding components based on their connections and whether they have conflicts. If a connection is free of conflicts, or the conflict is marked to be resolved by making the connection redundant the algorithm will continue the traversal through that connection. Otherwise the traversal will not go further. These conflicts are added according to a set of default rules or defined by the user. Every time the architect introduces a new conflict the traversal is repeated. Once the designer is satisfied with the components to be included and the duplication behavior introduced, the framework can be used to perform the actual modification of the architecture.

Every time a component is visited during the traversal, every connection is examined for existing redundancy and if it is found, they are marked as links with an existing redundancy conflict. Additionally, the rest of outputs are added an external output conflict if they connect one of the visited components with one that is not. Sometimes, the later component is visited later during the traversal, so the conflict does not apply anymore. An additional traversal detects them and mark to be resolved by making the connection redundant.

The user can define stop conflicts to determine the extent of the redundancy propagation. A stop conflict can be defined for a particular connection, or for a component, in this case it will apply to every connection belonging to the component. The architect can also include more components by solving existing redundancy conflicts as described before. Several resolution options are offered, namely merge "one to one", each "new one with each old one" or "duplicate existing redundancy".

IV. Evaluation

In order to demonstrate the proposed framework, it has been implemented into AirCADia Architect, a prototype software for systems architecting under the RFLP paradigm. An aircraft Environmental Control Systems (ECS), is analyzed form a safety perspective. The initial ECS architecture consists of the following views: requirements view (Fig. 3), functional view (Fig. 4) and logical view (Fig. 5). It features an ECS pack with a single heat exchanger which is fed high pressure air by an electrical compressor. The power for the compressor comes from one of the engine

electric generators. The ECS pack provides cold air and hot (trim) air. The former is mixed with recirculating air from the cabin at the manifold, this mix is further combined with trim air. The system is designed to accommodate this way the various atmospheric conditions in which the aircraft will flight. Since the focus is the safety of the architecture, proper performance of the architecture is assumed.

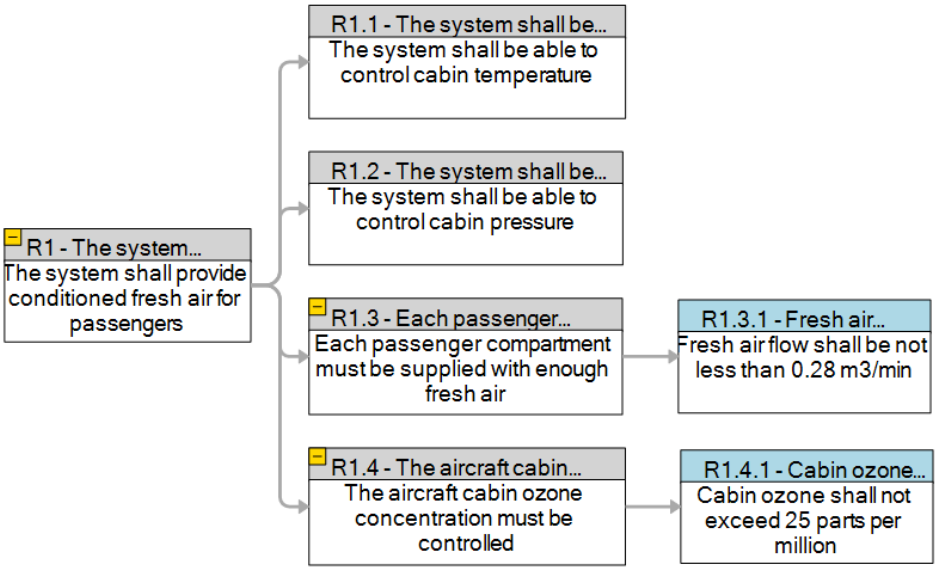

Fig. 3 ECS requirements view

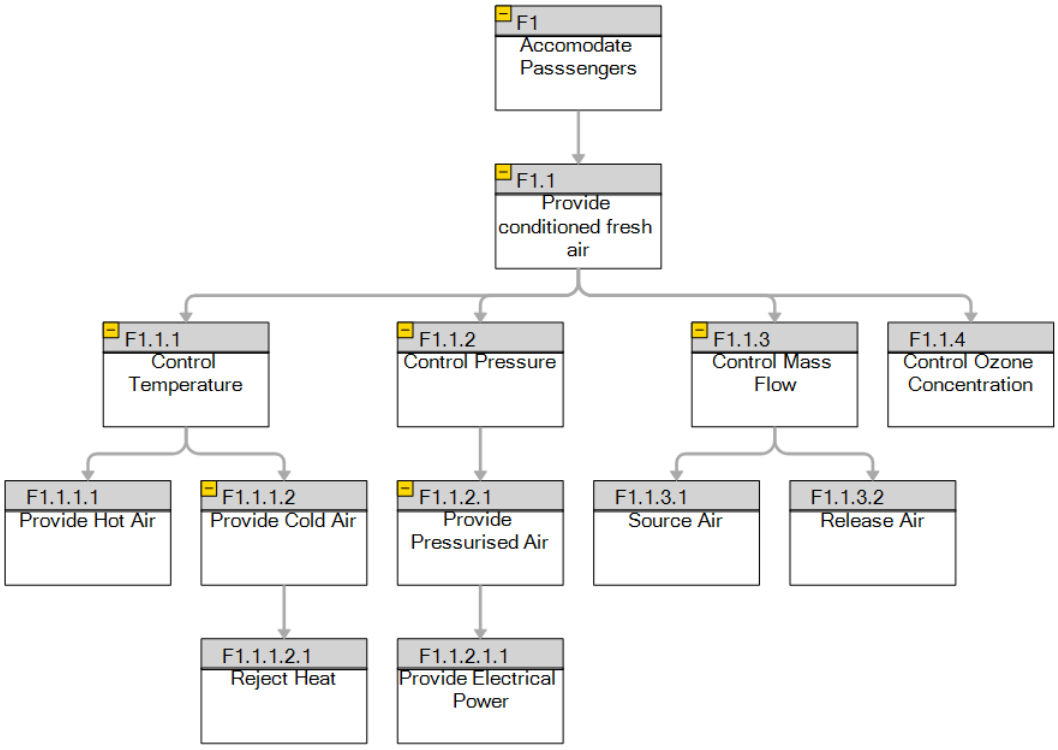

Fig. 4 ECS functional view

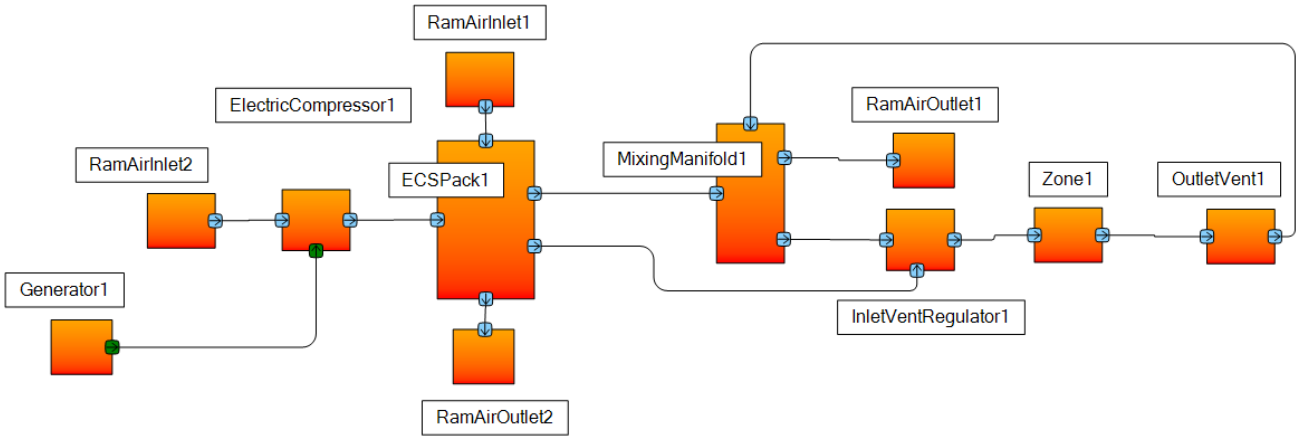

Fig. 5 ECS logical view

As it will be shown by the analysis, it fails to match some of the safety criteria. The proposed methodology is used to interactively locate the source(s) of the problem and to assist the architect into modifying the architecture to comply with safety requirements. Finally, the modified architecture is evaluated at to check if the changes have effectively solved the safety problems.

A. Functional Hazard Analysis

The first step is to determine the safety targets by performing a Functional Hazard Analysis of the architecture which is obtained from the functional view. The columns "Hazard Number" and "Function" in Table 2 are completed automatically. Although the rest must be completed by the user, which prevents the automation of an important part of the safety process, the FHA creation process still improves some aspects of the process. It enforces consistency between the architecture and the safety analysis. Additionally, it provides valuable help when determining the effects and causal factors of a hazard, as the functional-logical links of the architecture can be used to seamlessly navigate between FHA and the logical view, which possess relevant information about how elements are connected. An excerpt of the final results can be found in Table 2, where the gray colored text is the one added manually.

Table 2 – FHA results

ECS FHA					
Hazard Number	Function	Hazzard	Effect	Causal Factors	Classification
1	Control Mass Flow	Inability to Control Mass Flow	No right amount of new air can be provided to occupants.	Loss of capability to Source or Release Air	Catastrophic
2	Control Ozone Concentration	Inability to Control Ozone Concentration	Inadequate ozone concentration in air breathed by occupants	Loss of capability to Control Ozone Concentration	Hazardous
3	Control Temperature	Inability to Control Temperature	Inadequate temperature of air in cabin	Loss of capability to Provide hot or cold air	Major
4	…				

Once the classification of a failure is done, the affected requirements are updated using the hazards mapped to them. This relation is determined by the links between requirement and functional views, by using the graphs representing them [16], and the ones between functions and hazards, created with the FHA. For every requirement, all mapped hazards are examined, and the safety target displayed is the one with the strictest classification according to the FHA. It is possible to add them in quantitative or qualitative form. The updated requirements are shown in Fig. 6.

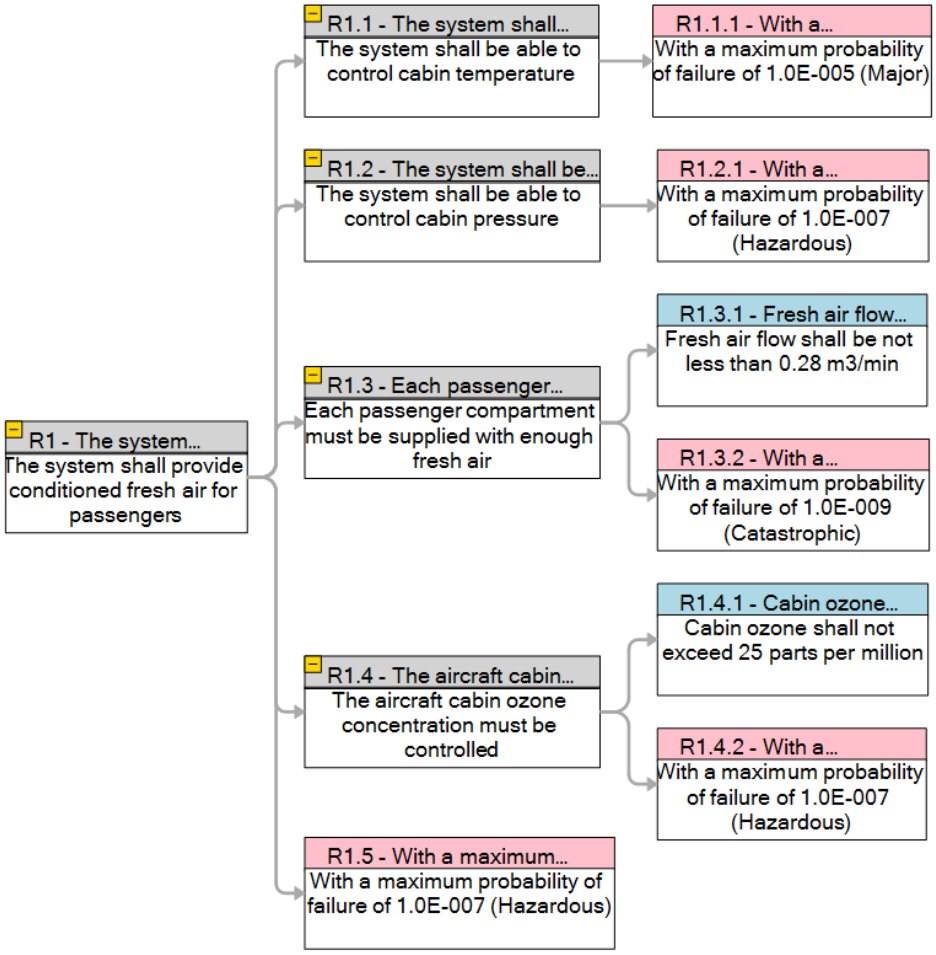

Fig. 6 Functions/Requirements traceability and safety requirements addition

B. Fault Tree Analysis

For those events that need qualitative or quantitative analysis (e. g. with severity equal or greater than Major) a fault tree can be created automatically using the information in the logical and functional views by using the fault tree algorithm describe in the previous section. If the probabilities of failure are included for all the components involved in the FTA it is possible to obtain quantitative results as well. For example, the Failure to control the cabin pressure leads to the tree in Fig. 7. The minimum cut sets are displayed in Table 3. As expected from the lack of redundancy of the architecture, every cut set is composed of only one item which generally indicates that the design of the architecture needs to be improved as the failure of only one component would prevent the system for performing the desired function. This is especially important in catastrophic failure conditions, for which certification requirements such as CS-25.1309(b)(1)(ii) impose that single failures must not exist.

Regarding quantitative results, the probability of the top event is displayed in the first row of Table 3. The safety of the system is not satisfactory since the probability of failure is greater than the target value. The minimal cuts are displayed in descending importance according to their relative probability (probability of the sets divided by the probability of the top event). The failure probabilities assigned to each element are for demonstration purposes. They are not real values, but they allow showing how different probabilities modify the rankings of components. All components have been assigned a probability of failure of 1e-10, except the ECS Pack, the Electric Compressor and

Generator which have been assigned 1e-5. As expected the greater contributor to the failure are the three elements with a higher failure probability. The Fussell-Vesley component ranking shows the same trend. The Birnbaum importance reflects the fact that every single failure causes the failure of the top event, all measures are one with the precision employed. However, with this precision no ranking information is provided by this measure. These elements are good candidates for modification in order to improve safety to acceptable levels.

Table 3 – FTA results

Probability of Failure: $P(\text{Failure to control the cabin pressure}) = 3 \cdot 10^{-5} > 10^{-7}$				
Minimal Cut Set	Number of Components	Relative Probability	Fussell-Vesley	Birnbaum
Generator	1	0.333	0.333	1.0
ECS Pack 1	1	0.333	0.333	1.0
Electric Compressor 1	1	0.333	0.333	1.0
Zone 1	1	3.33e-6	3.33e-6	1.0
Outlet Vent 1	1	3.33e-6	3.33e-6	1.0
Mixing Manifold 1	1	3.33e-6	3.33e-6	1.0
Ram Air Inlet 1	1	3.33e-6	3.33e-6	1.0
Ram Air Inlet 2	1	3.33e-6	3.33e-6	1.0
Inlet Vent Regulator 1	1	3.33e-6	3.33e-6	1.0

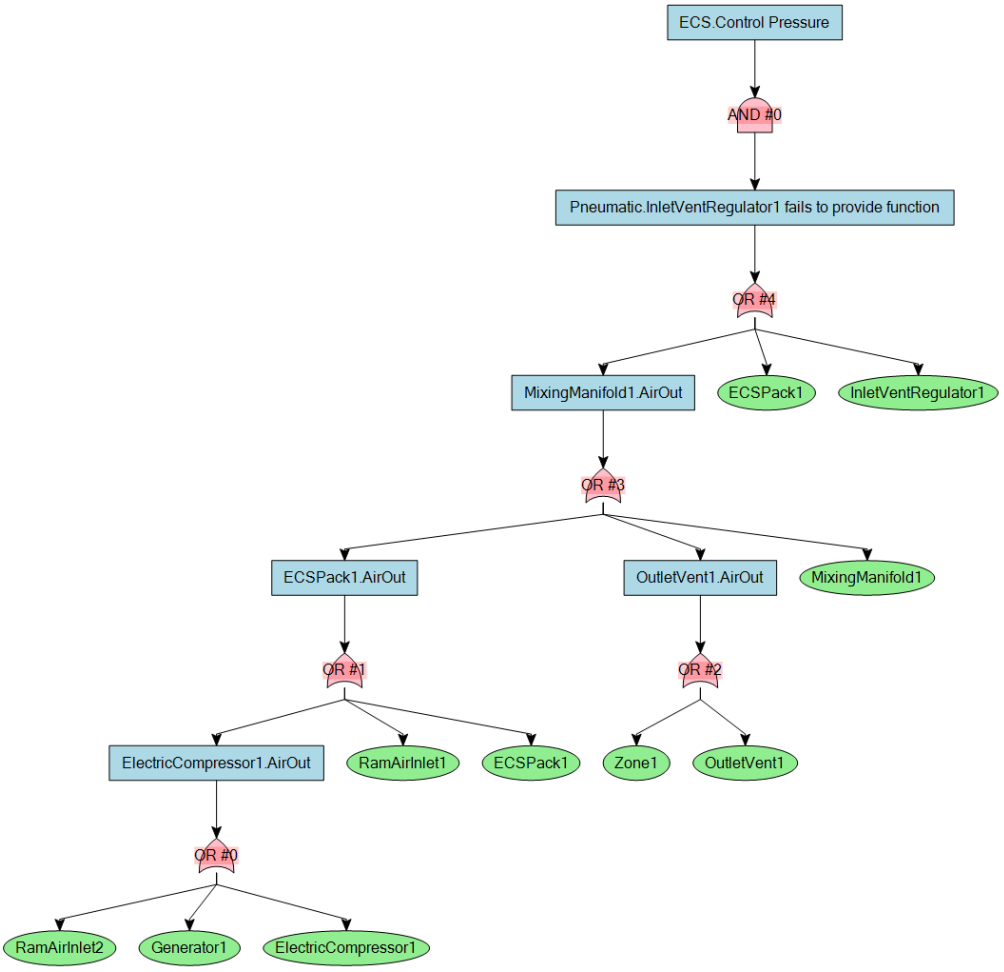

Fig. 7 Component-based and Function-based fault trees

C. Redundancy Addition

In order to improve safety, the effect of adding redundancy to the architecture is investigated. In a first stage, a redundant compressor is added. The first iteration of the algorithm selects for redundancy the compressor, the generator and the air inlet that feeds it, shown in blue in Fig. 8 Redundancy addition process Fig. 8a. It also detects one external output conflict regarding the connection with the ECS Pack, by default it is solved by duplicating the connection but not the ECS Pack (green connection in Fig. 8a). Since the probability of failure of the Ram Air Inlet is believed to be remote, it is decided to exclude it from redundancy by adding a user defined stop conflict, which is reflected as a green connection in Fig. 8b. Finally, the algorithm is ready to apply the redundancy as instructed. The final result corresponds to Fig. 8c, where the added elements are highlighted in yellow. Links with other elements of the architecture, such as function are also duplicated by the algorithm.

In a second stage, a redundant ECS Pack is added. The first iteration selects only the Pack itself as every output is marked by default as an output conflict as in the first stage and the inputs correspond to existing redundancy (marked in red in Fig. 9a), which also prevents the algorithm to continue the selection. In order to duplicate the Air Outlet, the default behavior is overridden, and the conflict resolution is changed to "make redundant". The existing redundancy conflict is solved by making redundant (again) the compressors. The algorithm reaches now every element on the left of the ECS Pack: electric compressors, generators and air inlet. Since the aircraft for which the system is being designed only has two engines, the generators are excluded so their final number is two. The final selection to be duplicated is shown in Fig. 9b. The resultant architecture, after applying redundancy can be seen in Fig. 9c.

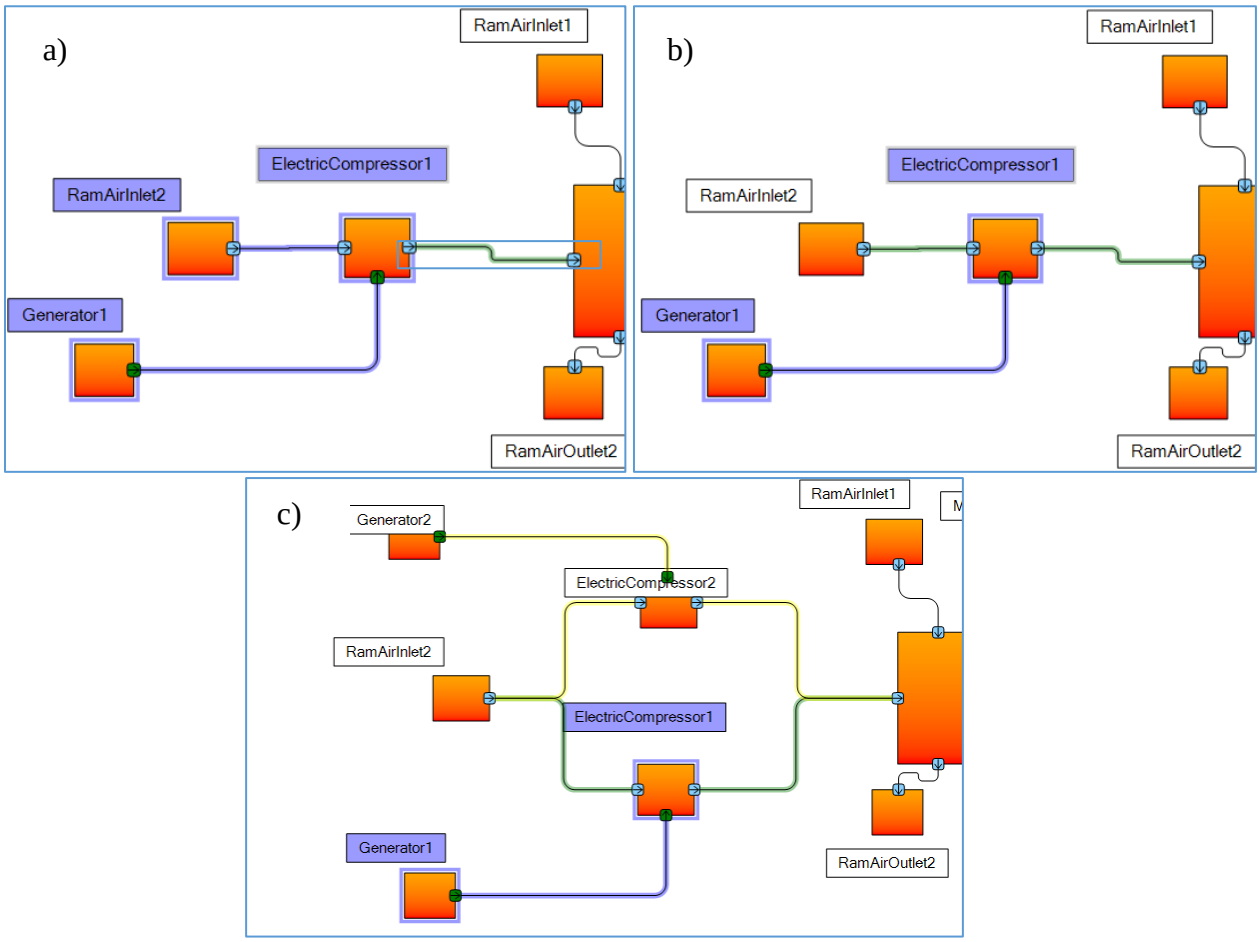

Fig. 8 Redundancy addition process. Stage 1

Finally, it is necessary to assess the improvement on the architecture safety, for this purpose fault trees are generated automatically from the new architecture. It can be observed from Fig. 10 that the topology of the fault tree for Failure to control the cabin pressure has changed substantially. In particular, the changes concentrate on the event that originally was labeled "ECSPack1.AirOut", and now correspond to "MixingManifold1.AirIn" whose child is an AND gate reflecting the redundant ECS packs. The redundant compressors also determine the presence of two more AND gates, one per each pair.

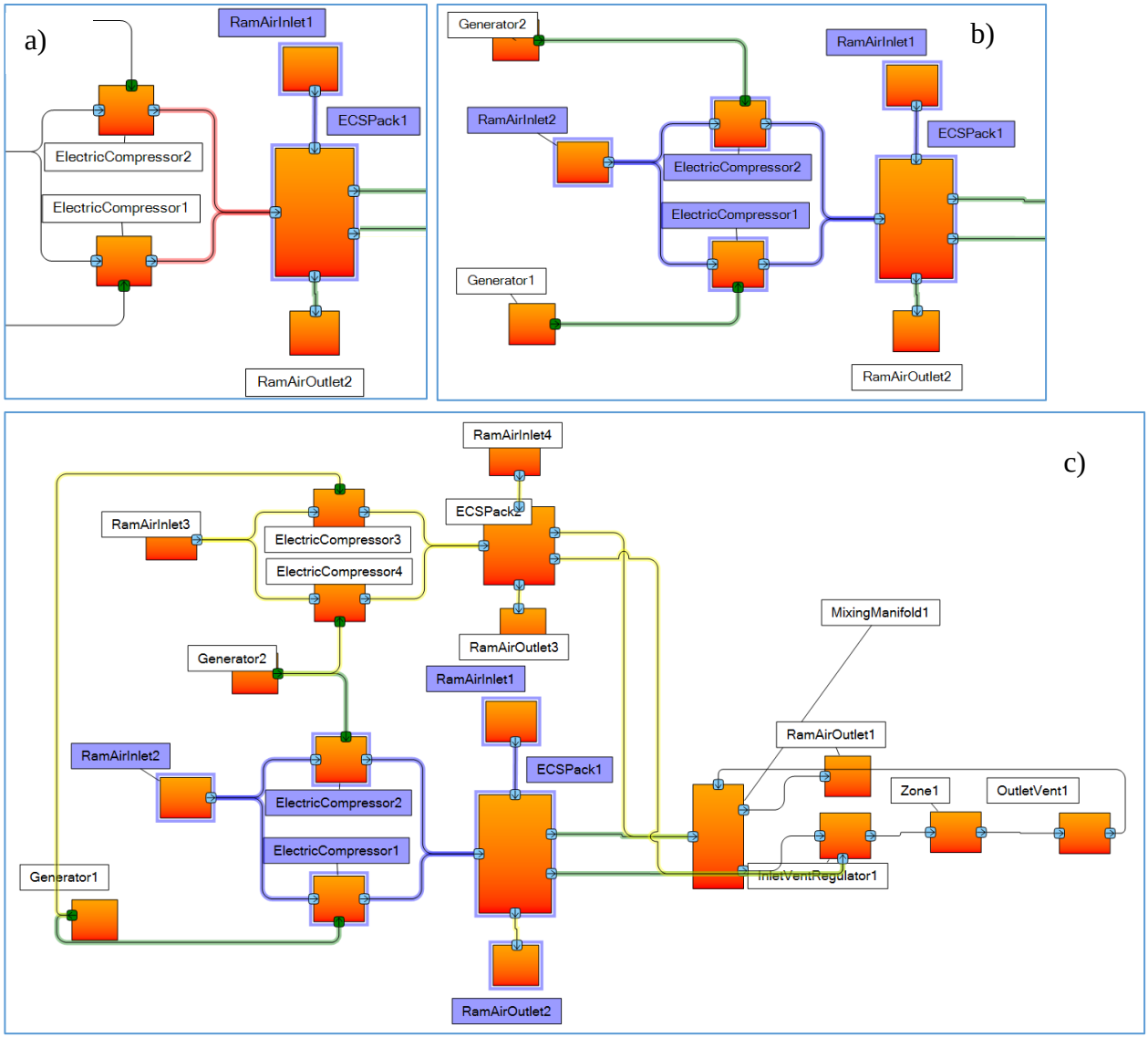

Fig. 9 Redundancy addition process. Stage 2

The new fault tree also presents different minimum cut sets (Table 4), where the single element cut sets have been reduced to four (the cardinality of the sets is displayed in the second column - N). The rest is ten sets of two elements, twenty sets of three and one set of four. Regarding their relative probabilities of failure, adding redundancy have shifted to lower values the contribution of the most unreliable components. Now they don't appear individually, and consequently the highest contribution is achieved when a combination of two of them causes the top event. Nevertheless, the contribution of them in pairs is only as important as that of the single element cut sets. The rest of combinations are from five to ten orders of magnitude smaller.

The diminution of the importance measures for redundant components is even clearer in the ranking in Table 5, where components are ordered by descending Fussell-Vesley importance. In particular the ECS Packs and generators have reduced their Fussell-Vesley importance down to the same level as more reliable non-redundant components

such as vents, mixing manifold or cabin zone. In case of the electric compressors, which present the greatest redundancy, this value is even lower, being close to the more reliable redundant air inlets. The Birnbaum measures show a different information. In this case, they seem to be correlated with the level of redundancy of the components. This seems to agree with its definition and the intuition that the more redundant a component is, the smaller is the impact of this component in the top event probability.

Regarding the overall impact of adding redundancy on the probability of the top event, it can be seen that the value has been improved in several orders of magnitude (6e-10). Now the architecture meets the safety requirement.

Table 4 – FTA results

Probability of Failure: $6 \cdot 10^{-10} > 10^{-7}$		
Minimal Cut Set	N	Relative Prob.
Inlet Vent Regulator 1	1	0.167
Zone 1	1	0.167
Outlet Vent 1	1	0.167
Mixing Manifold 1	1	0.167
Generator 1, Generator 2	2	0.167
ECS Pack1, ECS Pack2	2	0.167
ECS Pack1, Ram Air Inlet 3	2	1.67E-06
ECS Pack1, Ram Air Inlet 4	2	1.67E-06
ECS Pack2, Ram Air Inlet 1	2	1.67E-06
ECS Pack2, Ram Air Inlet 2	2	1.67E-06
Ram Air Inlet 1, Ram Air Inlet 3	2	1.67E-06
Ram Air Inlet 1, Ram Air Inlet 4	2	1.67E-06
Ram Air Inlet 2, Ram Air Inlet 3	2	1.67E-06
Ram Air Inlet 2, Ram Air Inlet 4	2	1.67E-06
Compressor 1, Compressor 2, Ram Air Inlet 3	3	1.67E-06
Compressor 1, Compressor 2, Ram Air Inlet 4	3	1.67E-06
Compressor 1, Compressor 2, ECS Pack 2	3	1.67E-06
Compressor 3, Compressor 4, Ram Air Inlet 1	3	1.67E-06
Compressor 3, Compressor 4, Ram Air Inlet 2	3	1.67E-11
Compressor 3, Compressor 4, ECS Pack 1	3	1.67E-11
Compressor 1, Compressor 3, Generator 2	3	1.67E-11
Compressor 2, Compressor 4, Generator 1	3	1.67E-11
Generator 1, Compressor 4, ECS Pack 1	3	1.67E-11
Generator 1, Compressor 2, ECS Pack 2	3	1.67E-11
Generator 1, Compressor 4, Ram Air Inlet 1	3	1.67E-11
Generator 1, Compressor 4, Ram Air Inlet 2	3	1.67E-11
Generator 1, Compressor 2, Ram Air Inlet 3	3	1.67E-11
Generator 1, Compressor 2, Ram Air Inlet 4	3	1.67E-11
Generator 2, Compressor 3, ECS Pack 1	3	1.67E-11
Generator 2, Compressor 1, ECS Pack 2	3	1.67E-11
Generator 2, Compressor 3, Ram Air Inlet 1	3	1.67E-11
Generator 2, Compressor 3, Ram Air Inlet 2	3	1.67E-11
Generator 2, Compressor 1, Ram Air Inlet 3	3	1.67E-11
Generator 2, Compressor 1, Ram Air Inlet 4	3	1.67E-11
Compressor 1, Compressor 2, Compressor 3, Compressor 4	4	1.67E-11

Table 5 – FTA results

Component	Fussell-Vesley	Birnbaum
ECS Pack 1	0.167	1.0E-5
ECS Pack 2	0.167	1.0E-5
Generator 1	0.167	1.0E-5
Generator 2	0.167	1.0E-5
Zone 1	0.167	1.0
Outlet Vent 1	0.167	1.0
Mixing Manifold 1	0.167	1.0
Inlet Vent Regulator 1	0.167	1.0
Electric Compressor 1	5.00E-6	1.0E-10
Electric Compressor 2	5.00E-6	1.0E-10
Electric Compressor 3	5.00E-6	1.0E-10
Electric Compressor 4	5.00E-6	1.0E-10
Ram Air Inlet 1	1.67E-6	1.0E-5
Ram Air Inlet 2	1.67E-6	1.0E-5
Ram Air Inlet 3	1.67E-6	1.0E-5
Ram Air Inlet 4	1.67E-6	1.0E-5

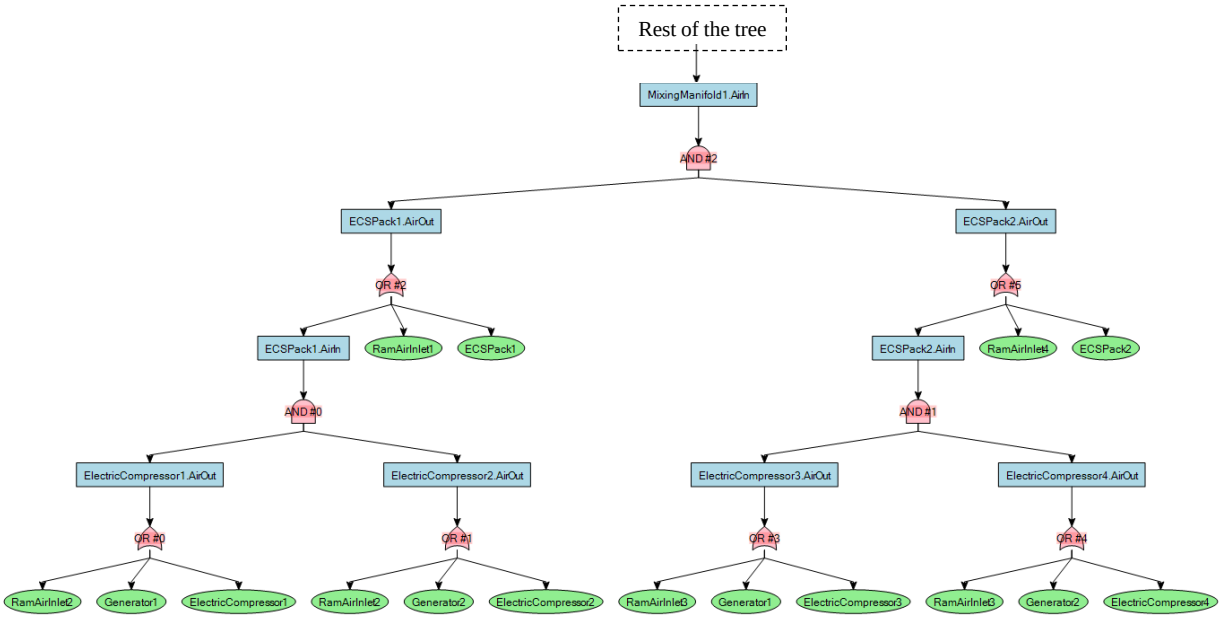

Fig. 10 Detail of the fault tree of the redundant architecture

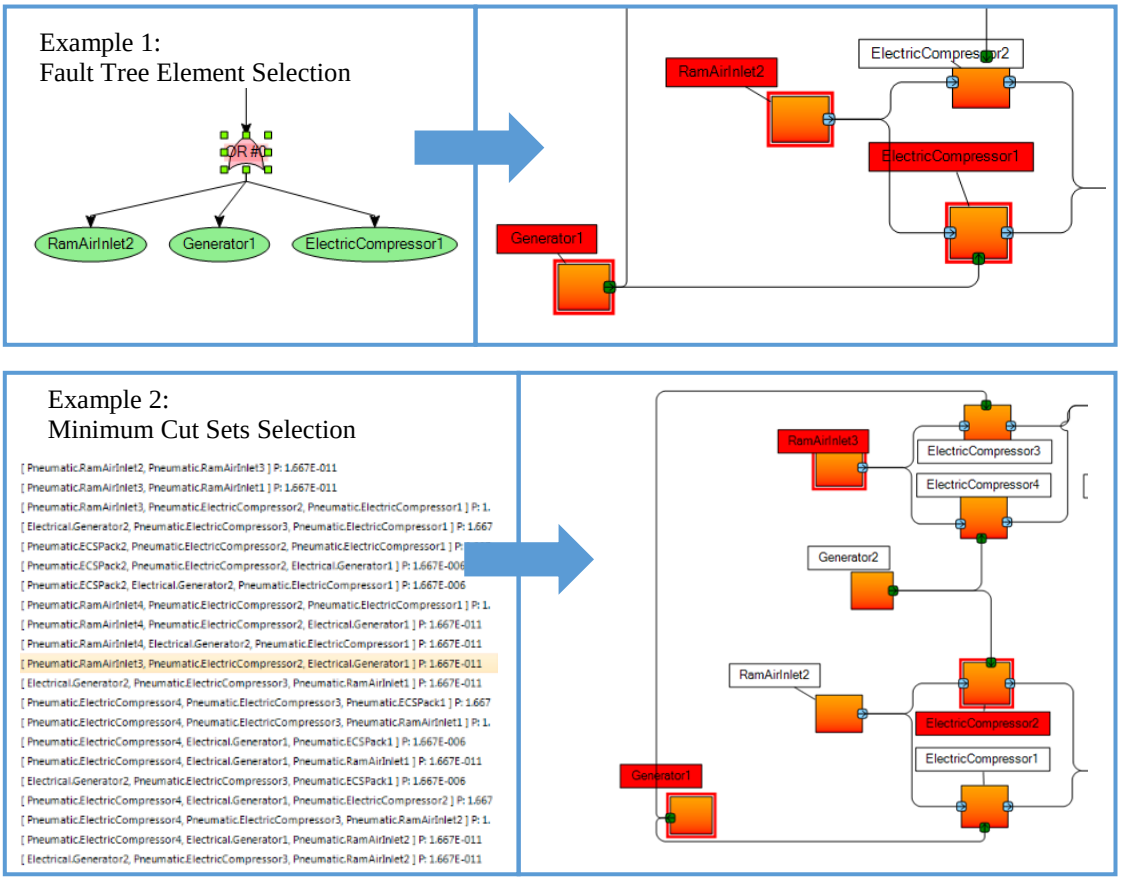

Fig. 11 Example of interactivity enablers

D. Interactivity enablers

Finally, the interactive capabilities are shown through two examples in Fig. 11. The first one shows how the leftmost gate on the fault tree in Fig. 10 maps to the respective components in the logical view. This link has been created automatically at the same time than the fault tree. Similarly, example two shows how the cut set containing the elements Generator 1, Compressor 2 and ECS Pack 2 maps to their representation in the logical view. This way, the effort of locating the components while interpreting FTA results is greatly decreased.

V. Conclusions and future work

Presented is a framework for incorporating safety analysis in early systems architecture design. It utilizes the Requirement, Functional, Logical, and Physical (RFLP) paradigm, augmented with a Computational domain for automated systems sizing. An algorithm to support partial automation of the Functional Hazard Analysis process, using the Functional View as input, is described. The FHA results are then used to update the safety objectives in the requirements view. Another algorithm is introduced to automatically generate Fault trees are from the Logical view. These trees can be evaluated both qualitatively and quantitatively and the contribution to the probability of failure of their components can be ranked through importance measures, helping the architect to focus on the most problematic parts of the architecture. Finally, and interactive approach to introducing redundancy in the architecture is proposed. The application of the framework is illustrated with a representative example.

The algorithms require only minimum modification of the RFLP views definitions: addition of probabilities of failure for logical components and redundancy type for logical connections. This has the advantage that automatic/interactive safety assessment can be applied easily to any RFLP existing architecture definition. However, this introduces some limitations when describing more complex fault propagation scenarios and component with different states. Incorporating these advanced scenarios is planned for future work.

Also, when the system is modified to comply with the safety requirements, the performance of the aircraft might change depending on the scale of the changes. For example, the added components might add weight, and this could affect the sizing of other components and consequently the performance at aircraft level. Work is already underway to adapt the method proposed by Bile et al. [18] for (semi)automated sizing and system performance assessment after safety-related modifications.

References

[1] EASA, "Certification Specifications and Acceptable Means of Compliance for Large Aeroplanes CS-25," no. Ammendment 20, 2017.

[2] A. Joshi, S. Vestal, and P. Binns, "Automatic Generation of Static Fault Trees from AADL Models," *DSN Work. Archit. Dependable Syst.*, 2007.

[3] F. Mhenni, N. Nguyen, and J.-Y. Choley, "Automatic fault tree generation from SysML system models," in *2014 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, 2014, pp. 715–720, doi:10.1109/aim.2014.6878163.

[4] M. Roth, M. Wolf, and U. Lindemann, "Integrated Matrix-based Fault Tree Generation and Evaluation," *Procedia Comput. Sci.*, vol. 44, pp. 599–608, Jan. 2015, doi:10.1016/j.procs.2015.03.027.

[5] J. Xiang, K. Yanoo, Y. Maeno, and K. Tadano, "Automatic Synthesis of Static Fault Trees from System Models," in *2011 Fifth International Conference on Secure Software Integration and Reliability Improvement*, 2011, pp. 127–136, doi:10.1109/ssiri.2011.32.

[6] Y. Papadopoulos, J. McDermid, R. Sasse, and G. Heiner, "Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure," *Reliab. Eng. Syst. Saf.*, vol. 71, no. 3, pp. 229–247, Mar. 2001, doi:10.1016/s0951-8320(00)00076-4.

[7] J. Delange and P. Feiler, "Architecture Fault Modeling with the AADL Error-Model Annex," in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, 2014, pp. 361–368, doi:10.1109/seaa.2014.20.

[8] S. Li and X. Li, "Study on Generation of Fault Trees from Altarica Models," *Procedia Eng.*, vol. 80, pp. 140–152, Jan. 2014, doi:10.1016/j.proeng.2014.09.070.

[9] A. Majdara and T. Wakabayashi, "Component-based modeling of systems for automated fault tree generation," *Reliab. Eng. Syst. Saf.*, vol. 94, no. 6, pp. 1076–1086, Jun. 2009, doi:10.1016/j.ress.2008.12.003.

[10] G. Latif-Shabgahi and F. Tajarrod, "A New Approach for the Construction of Fault Trees from System Simulink," in *2009 International Conference on Availability, Reliability and Security*, 2009, pp. 712–717, doi:10.1109/ares.2009.172.

[11] Y. Papadopoulos and M. Maruhn, "Model-based synthesis of fault trees from Matlab-Simulink models," in *Proceedings International Conference on Dependable Systems and Networks*, pp. 77–82, doi:10.1109/dsn.2001.941393.

[12] C. Schallert, "Inclusion of Reliability and Safety Analysis Methods in Modelica," in *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany*, 2011, no. 63, pp. 616–627, doi:10.3384/ecp11063616.

[13] "ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and

Equipment," 1996.
[14] E. Ruijters and M. Stoelinga, "Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools," *Comput. Sci. Rev.*, vol. 15–16, pp. 29–62, Feb. 2015, doi:10.1016/j.cosrev.2015.03.001.
[15] Y. Dutuit and A. Rauzy, "Efficient algorithms to assess component and gate importance in fault tree analysis," *Reliab. Eng. Syst. Saf.*, vol. 72, no. 2, pp. 213–222, May 2001, doi:10.1016/s0951-8320(01)00004-7.
[16] M. D. Guenov, A. Riaz, Y. Bile, A. Molina-Cristóbal, and A. S. J. van Heerden, "Information System Support for Aerospace Vehicle Systems Architecting," *J. Aerosp. Inf. Syst.*, 2018.
[17] VDI, *Design methodology for mechatronic systems (VDI 2206)*. Verein Deutscher Ingenieure, 2004.
[18] Y. Bile, A. Riaz, M. D. Guenov, and A. Molina-Cristobal, "Towards Automating the Sizing Process in Conceptual (Airframe) Systems Architecting," in *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, 2018, pp. 1–16, doi:10.2514/6.2018-1067.
[19] J. von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms From Unreliable Components," in *Automata Studies. (AM-34)*, Princeton: Princeton University Press, 1956.
[20] F. P. Mathur and P. T. de Sousa, "Reliability Modeling and Analysis of General Modular Redundant Systems," *IEEE Trans. Reliab.*, vol. R-24, no. 5, pp. 296–299, Dec. 1975, doi:10.1109/tr.1975.5214914.
[21] D. E. Eckhardt *et al.*, "An experimental evaluation of software redundancy as a strategy for improving reliability," *IEEE Trans. Softw. Eng.*, vol. 17, no. 7, pp. 692–702, Jul. 1991, doi:10.1109/32.83905.
[22] M. Ian and A. Seabridge, *Design and development of aircraft systems*, 3rd ed. West Sussex, England: John Wiley & Sons, 2012.
[23] C. A. Ericson and others, *Hazard analysis techniques for system safety*. John Wiley & Sons, 2015.
[24] P. J. Wilkinson and T. P. Kelly, "Functional hazard analysis for highly integrated aerospace systems," in *IEE Certification of Ground/Air Systems Seminar*, 1998, pp. 4–4, doi:10.1049/ic:19980312.
[25] Y. Bile, "Yogesh´s Thesis," Cranfield University, 2018.
[26] J. B. Fussel, E. B. Henry, and N. H. Marshall, "MOCUS a Computer Program to Obtain Minimal Sets From Fault Trees," *ANCR Math. Comput.*, vol. 55, no. 2, pp. 51–55, 1974.
[27] D. Kececioglu, *Reliability engineering handbook*, vol. 2. DEStech Publications, Inc, 2002.
[28] A. Rauzy, "Toward an efficient implementation of the MOCUS algorithm," *IEEE Trans. Reliab.*, vol. 52, no. 2, pp. 175–180, Jun. 2003, doi:10.1109/tr.2003.813160.
[29] U. Lindemann, M. Maurer, and T. Braun, "The procedure of structural complexity management," in *Structural Complexity Management*, Berlin, Heidelberg, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 61–66.
[30] J. A. McDermid, "Support for safety cases and safety arguments using SAM," *Reliab. Eng. Syst. Saf.*, vol. 43, no. 2, pp. 111–127, Jan. 1994, doi:10.1016/0951-8320(94)90057-4.
[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2010.