Systems Engineering,Volume 23, Issue 3, May 2020, pp. 350-365
DOI: 10.1002/sys.21531

ORIGINAL ARTICLE

Journal Section

# Computational Framework for Interactive Architecting of Complex Systems

**Marin D. Guenov[1] | Atif Riaz[1] | Yogesh Bile[1] | Arturo Molina-Cristobal[1] | Albert S.J. van Heerden[1]**

[1]Centre for Aeronautics, Cranfield University, Bedford, Bedfordshire, MK43 0AL, United Kingdom

**Correspondence**
Marin D. Guenov, Centre for Aeronautics, Cranfield University, Bedford, Bedfordshire, Postal MK43 0AL, United Kingdom
Email: m.d.guenov@cranfield.ac.uk

**Funding information**
European Union Seventh Framework Programme (FP7/2013-2016, TOICA project), Funder One Department, Grant Number: 604981

Presented is a novel framework for interactive systems architecture definition at early design stages. It incorporates graph theoretic data structures, entity relationships and algorithms which enable the systems architect to operate interactively and simultaneously in different domains. It explicitly captures the 'zigzagging' of the functional reasoning process, including not only allocated, but also the derived functions. A prototype software tool, AirCADia Architect, was implemented, which allowed the framework to be demonstrated to and tried hands-on by practicing aircraft systems architects. The tool enables architects to effectively express their ideas when interactively synthesizing new architectures, while still retaining control over the process. The proposed approach was especially acknowledged as the way forward for rationale capture.

**KEYWORDS**
Systems Engineering, Architecture Definition

## 1 | INTRODUCTION

The ability to innovate is considered one of the key factors for success in the globally competitive world of today. This is particularly relevant in the design of complex systems, such as aircraft, where the requisite subsystems, for example, Environmental Control, Flight Control, etc., account for about one-third of the total empty weight and cost [1]. Innovation in aircraft subsystems design can bring forth significant competitive advantages, such as improved fuel consumption, reduced maintenance costs, and higher reliability.

The work reported in this paper is related to innovative systems architecting and originates from a topical European research project, "Thermal Overall Integrated Conception of Aircraft" (TOICA) [2] [3]. Specifically, the authors contributed to one of the research objectives of TOICA: the development of methods and tools enabling systems architects to discover, define, assess, and evaluate (systems) architectures. Within this wider context, the scope of the research described herein has been restricted to the process of conceptual systems architecture definition, within the requirements (R), functional (F) and logical (L) domains. The term requirements domain refers to the activities involving requirements engineering and management. The functional domain is concerned with the functional decomposition and analysis, and the logical domain deals with the specification of solutions and their interfaces. In the systems engineering community, solutions are also referred to as forms, means or artifacts, and include both components and sub-systems. Also, logical solutions are sometimes referred to as physical solutions. In this paper, we shall distinguish between logical and physical domains, where the former is assumed to be concerned mainly with the interconnectivity between components and subsystems, while the latter with spatial layout and geometry. In a recent publication [4], we presented data structures underpinning basic operations which take place in the F-L domains. Here we extend that work to include requirements and the notion of a computational (behavioral) domain. The latter is intended to facilitate the fast (steady state) assessment of a rapidly synthesized architectures, which is important for design space exploration at precompetitive or early product development stages. The details of the sizing approach are reported in [5]. Here the computational domain is used specifically as a source of information needed to identify which functions and solutions are affected if a performance requirement is modified.

The rest of the paper is organized as follows. Basic concepts and terminology are introduced in Section 2 with the help of a case study. State-of-the-art methods and tools, and associated research challenges are identified in Section 3. The proposed components of the framework, underpinning the functional reasoning and inter-domain traceability are described in Section 4. The framework, implemented in a prototype software tool, is demonstrated through an application case study in Section 5. Finally, conclusions, current limitations and recommendations for future work are outlined in Section 6.

## 2 | BACKGROUND

Systems architecting is the process of creating and describing complex systems architectures. Crawley et al. define system architecture as "the embodiment of concept, the allocation of function to elements of form, and definition of relationships among the elements and with the surrounding context" [6]. Therefore, the main activity performed during systems architecting is making decisions about the system concept, i.e. which functions are to be achieved, which solutions are to be employed in order to realize those functions, and what are the relationships between these elements. These architectural decisions will inevitably influence the performance characteristics of the system.

Architecting and engineering are often seen as two different activities, and the differentiation between the two is often contentious. According to Maier [7], architecting or "architecture definition" deals largely with unmeasurable entities using non-quantitative tools and methods such as decomposition of requirements and functions, allocation of solutions to functions, and specification of interfaces between solutions. By contrast, the term "architecture assessment" refers to engineering, i.e., activities involving measurable quantities, such as component sizing and optimization. In practice, the two activities are related and are performed iteratively. As mentioned above, the work reported in this paper focuses on architecture definition (architecting). The rest of this section outlines a case study, which is used to define some of the terminology used in the paper.

Suppose the top-level requirements related to the environmental control system of a transport aircraft are al-

ready specified, as shown for illustration purposes only at the top of Figure 1. (In practice, requirements elicitation and management tools such as IBM Rational DOORS [8] can be used). During the functional design, the requirements are mapped to functions, which are then decomposed into sub-functions. For example, a top-level requirement such as “The system shall provide comfortable environment for passengers” is first mapped to a function "Provide comfortable environment for passengers". This, in turn, is decomposed into four sub-functions: "source air", "circulate air", "cool air" and "evacuate hot air". Next, the systems architect needs to consider how to source air. At this level of reasoning, he/she identifies two possible solutions, "ram air inlet" and "pneumatic system". Assuming the former, the atmospheric air needs to be pressurized. Subsequently, if the architect selects a compressor to fulfill this function, then a *derived function* "provide rotational energy" needs to be defined. In Figure 1, the derived functions are linked through dashed lines in the functional decomposition. It can be observed that the sourced air goes through a number of transformations, implying a (process) flow, which can be inferred in one step, before even specifying equipment connections, or in a “zig-zagging” fashion, as shown in the bottom-right box of Figure 1. It is important to note that the functions in the decomposition hierarchy usually depend on the solutions selected at a higher level. Similar reasoning can be applied should the architect have chosen a pneumatic system to fulfill (implement) the function source air. Note that the derived functions (Convert Ozone and Demist Air) are connected through dashed lines in Figure 1. That is, they are derived from the chosen means (solution) for sourcing air. It can be observed, on the whole, that the architect has to deal with the functional decomposition, the identification of possible functional flows and the mapping of these functions to solutions (components or subsystems).

## 3 | STATE OF THE ART

Model-based systems engineering (MBSE) is "the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases" [9]. A considerable body of work on formal methods and principles of systems engineering has been published over the last several decades. Amongst the pioneers, Hall [10] suggested a description of high-level functions and activities to manage the systems engineering process. Wymore [11] proposed two types of mathematical models: 1) System models, based on physics, drawings and specifications, and 2) a theoretical model for management of personnel and resources (hardware/software) throughout the life cycle of the system. The theoretical model utilizes a system design language based on set theory. Micouin [12] introduced a formal model-based systems engineering methodology called Property-Model Methodology. It is a sequence of operative rules, defined to build specification and design models of engineered systems and is expressed in a language. More recently, Luzeaux [13] proposed the application of category theory as a formal foundation of systems engineering, claiming that it is better adapted to current systems modelling tools and languages. These mathematical models for systems engineering are limited in that they do not account for the interactive ‘zigzagging’ process involved in performing functional reasoning. For example, the Systems Coupling Recipe (SCR) concept [11] assumes that the connectivity is given, that is, the decomposition and connectivity have been worked out in advance. By contrast our work aims to facilitate the decomposition process which results in functional and logical (connectivity) views of the system. Models such as the SCR can be extracted from these for formal testing of the system. With this regard, one of the intentions behind the proposed framework is to extend the existing mathematical models by employing decomposition relations, including derived functions.

There are many modelling languages for defining architectures of complex systems. Notable examples include the Systems Modeling Language (SysML) [14], Object Process Methodology (OPM) [15], STRATA [16], Lifecycle Mod-

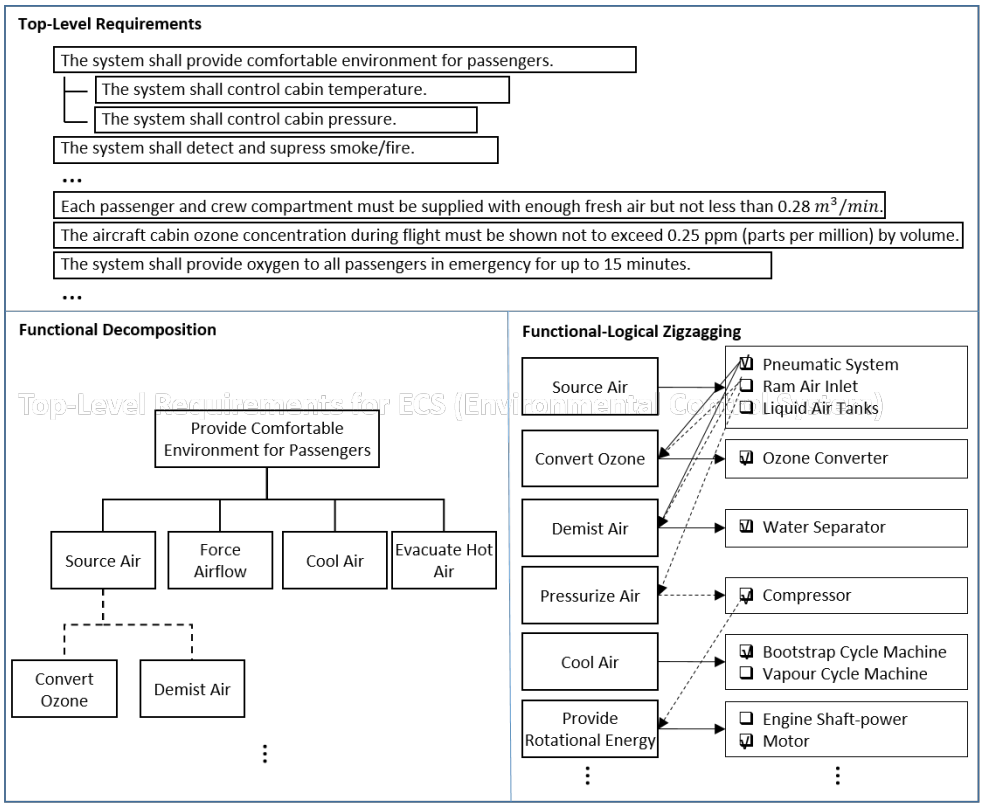


**FIGURE 1** Top-level requirements, functional decomposition, and functional-logical zigzagging

eling Language (LML) [17], and the United States Department of Defense Architecture Framework (DoDAF) Meta Model (DM2) [18]. These languages use both graphical and textual means to document aspects of the architecture. For example, the DoDAF Meta Model (DM2) combines logical constructs with an ontology to capture and define a vocabulary for the description of DoDAF models and their usage in core processes such as operation planning, budgeting, acquisition, and so forth. SysML uses object-oriented design constructs and a limited ontology to capture the systems description. LML utilizes both simplified constructs and ontology to enable the sharing of cost, schedule and performance data between all stakeholders in the system lifecycle. The original purpose behind LML was to replace predecessors, such as SysML which was seen as overcomplicating the systems engineering process.

The aforementioned modelling languages are intended primarily to describe the synthesized architecture, but are less suited to capturing the actual architecting process. For instance, if the architect wishes to create a number of new candidate architectures by modifying and/or combining existing ones, it will be very tedious to identify all the affected elements (requirements, functions, solutions, etc.). Thus, there is a need for a framework that can enable the architect to interactively define and explore multiple architectures at the early design stages. With this regard, instead of defining a new modelling language, the present work was focused on enabling the functional reasoning and decomposition process. That is, determining the functions that the product must perform, the interactive co-evolution of the functional-logical domain and capturing the actual process of architecting a system.

Functional reasoning appears to be a central tenet in modern systems engineering practice. It is believed that this

approach discourages the designers from immediately elaborating on the first solution that comes into mind, which may not be the best. The standards for systems engineering processes, such as EIA 632 [19] and ISO/IEC 15288 [20], also prescribe that (functional) requirements are developed in a solution neutral environment to allow the exploration of different solutions (physical embodiments).

Several distinct functional reasoning models have been developed and reported in the academic literature. A few of these, e.g., Functional Decomposition [21] and Functional Basis [22] [23] support functional description in a solution neutral way and in the form of a verb-object pair, e.g., "source air". Others, e.g., the Freeman and Newell's model [24], Chakrabarti's functional reasoning model [25] and the Function Behaviour State (FBS) approach [26] allow functional description with reference to the solution under consideration. A related methodology, which advocates the "zig-zagging" decomposition described in the previous section is Axiomatic Design (AD) [27] [28].

Functional reasoning as part of the systems architecting process can be thought of as distributed over four notional domains: Requirements, Functional, Logical and Physical – referred to as 'RFLP'. Specifically, RFLP stands for Requirements engineering, Functional design, Logical design, and Physical design and describes the process of systematic product development, from system analysis to system development [29]. RFLP is based on the German guideline VDI 2206, "Design methodology for mechatronic systems" [30], quoted also in [31]. RFLP has gained popularity, not least because of its adoption by leading product lifecycle management (PLM) vendors [32], [33]. It appears that, in its current application, RFLP is unidirectional. Indeed, as mentioned above, the prevailing opinion in the engineering design field is still that form-follows-function. That is, the functional decomposition is followed by a logical and physical product decomposition (mapping) which aligns with systems thinking. However, a review of recent cognitive and organizational sciences literature related to creativity in design (e.g. Refs. [34] [35] [36]) suggests that, in many cases, the opposite is true. For example, experiments, as reported in Ref. [36], indicate that solutions provided under the more structured function-follows-form condition would be judged more original and creative than those provided under the less structured form-follows-function condition and also that solutions provided by intuitive participants (in the experiment) would be judged more original and creative than those provided by systematic participants, but mainly in free (less structured) conditions. This is related to design thinking. Additionally, Vermaas [37] has shown from a philosophical point of view, that the relation between technical functions and their sub-functions in (abstract) functional descriptions of technical products cannot be analyzed as a formal relation of parthood. That is, operating solely with (abstract) functional decompositions may lead to the paradox of a function containing an instance of itself. This appears to be an additional argument for the interactive co-evolution of the functional and logical domains in practice. Last, but not least, there is a view in the field of Psychology that "a clear, unequivocal, and incontestable answer to the question of how creativity can be enhanced is not to be found in the psychological literature" (Ref. [38], p. 407, cited in Ref. [39]).

It can be seen from the review of systems architecting models in the academic literature and in practice, that there is no consensus on the formalisms and the creative process of synthesizing architectures and whether design and systems thinking are that different at their core [40]. While adopting the RFLP notion and the sequential-iterative nature of the systems engineering process, as stipulated in existing standards, we concluded that we ought to research methods and tools, which when combined into an innovative framework would enable the systems architects to work interactively and simultaneously in all domains. This requires that the computer aided architecting system maintains background traceability of the architect's choices in the RFLP domains during the process. In the next section, the basic components of such a framework are specified.

# 4 | FRAMEWORK SPECIFICATION

This section describes the proposed systems architecting framework which supports the RFLP paradigm and in particular, the interactive (architecting) operations in the requirements, functional, logical and computational domains.

## 4.1 | Basic Architectural Elements and Object Model

The basic classes of the framework and their relationships are shown in Figure 2. The object model serves as the foundation of the prototype tool, AirCADia Architect, used for the demonstration (see Section 5) of the proposed approach. It has to be emphasized that in practice, information related to some of the classes may be part of an organization's PLM system, including requirements management.

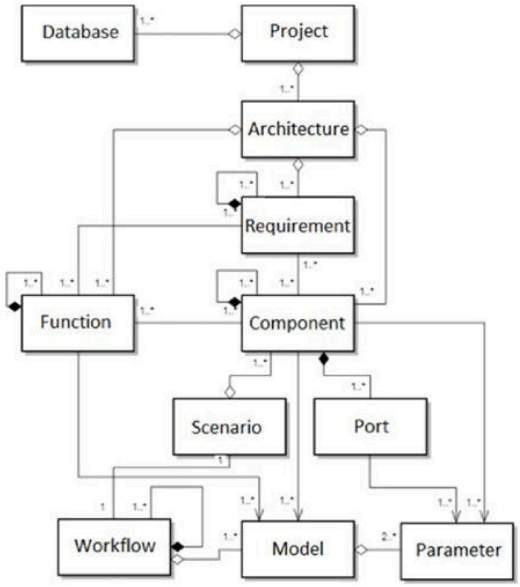


**FIGURE 2** UML class diagram of the proposed framework (attributes and methods are not shown)

The architecture class serves as a container for storing different elements (requirements, functions, solutions) from all domains. The 'Database' class stores knowledge about all the architectural elements (including requirements, functions, solutions, computational models, etc.) defined by the architect. The 'Project' class is a container for both the database object and the list of architectures. It is a utility class, which is used for storing studies as part of the software implementation of the framework.

The 'Requirement' class represents the technical requirements transformed from the stakeholders' needs, which could be of type functional or performance. It has attributes and methods enabling the requirements decomposition process, and the requirement-to-function mapping relations. Similarly, the 'Function' class represents a function and has attributes and methods enabling the functional decomposition process. A function object, $\varphi$ is the action that a product or system has to perform in order to meet the stakeholders' needs. Functions are expressed as a 'verb-noun' combination, $\varphi(n)$, where '$\varphi$' is the action and '$n$' is the object of the action, e.g., dehumidify (air). For convenience,

the object, '*n*', will be omitted from now on, unless explicitly required. A function stores a reference to a component (and vice-versa) to implement the concept of function-solution mapping and derived functions, respectively.

The 'Component' class stands for physical solution(s) satisfying the functions. A component, $\sigma$, is the physical element or elements (e.g. part, component, subsystem, or even the whole system/product) that performs (fulfills) the required function. A component may also have attributes that represent component's parameters (e.g. an electrical transformer has transformation ratio and efficiency coefficient). The interface parameters of the component (e.g. in case of the transformer, input/output voltage and current) are managed by port, which are described below. Similar to the function class, the 'Component' class has attributes and methods enabling the function-solution mapping and the decomposition process.

The 'Port' class describes the interfaces of a component with other components and the environment, including the type of the flow passing through the interface (e.g. energy, material, signal, etc.) and the direction, which can be either 'input' or 'output'. The Port class may also contain a description of flow parameters (e.g. in the case of an electrical interface, 'voltage' and 'current'). Such parameters provide an additional level of compatibility control between components or solutions. Another important reason to introduce the port class is that it is intended to facilitate the export of the synthesized architectures to sizing and analysis tools (e.g. Modelica).

The 'Parameter' class represents an engineering quantity that describes some characteristics of a solution or a port in the logical domain. For instance, compression ratio is an example of a parameter associated to the compressor solution. Similarly, both the input and output ports of the compressor may have two associated parameters, pressure and air mass flow rate. Two types of parameters are considered: components' direct parameters (e.g. compression ratio of the compressor), and components' ports parameters (e.g. pressure of fluid at input/output port of the compressor).

The 'Model' class represents the computational code (mathematical equations) for predicting the solutions' behavior or performance characteristics. Each computational model has one or more output parameters and the quantities of these output parameters are calculated using the given quantities of one or more input parameters. Depending on context, each solution in the logical domain has two types of behaviors: intended and unintended. For instance, in the case of an electrical motor, the intended behavior could be to provide torque, while the 'unintended' behaviors may be generation of heat and vibration. It is vital that unintended behaviors are considered during architecting, because these may impose modifications on the envisaged architecture. For instance, an electric motor based electro-mechanical actuator (EMA) of a flight control system may necessitate the use of dedicated thermal management solutions, such as heat pipes, if the natural radiation and convection is not sufficient to keep the EMA at the acceptable operating temperature.

The 'Workflow' class represents the network of computational models used for sizing the system and describes the execution sequence of the computational models. A computational model is connected to other computational models through shared parameters (variables). The collection of parameters, models, and workflow constitute a virtual domain, which is referred to as the Computational domain. In the context of this work, the computational domain is especially useful for tracing variable dependencies between different product decomposition levels.

It is important to note that the architects are not required to interact directly with the computational domain; they only need to specify the required parameters and computational models (which are developed by simulation specialists) for behaviors and performance characteristics. Automatic (dynamic) workflow creation methods [41] [5] can be employed to generate the computational workflows 'behind the scene'.

## 4.2 | Elementary Relations

As stated earlier, the aim of this work is not to prescribe a rigid requirements decomposition process, but to specify essential R-F-L mappings. The latter are intended to allow the application of formal graph theoretic structures (described later) that captures the functional reasoning process and the evolving architecture. The basic relations between architectural elements employed in the proposed framework are presented below:

**Requirement-to-Function and Requirement-to-Parameter:** This relation specifies the mapping from a functional requirement to a function, i.e. $\rho_1 \mapsto \varphi_1$. Similarly, there is a relation from the performance requirement (constraint) to a solution's parameter, $\rho_1 \mapsto p_1$. Multiple mapping relations can be specified, e.g. a single functional requirement may be mapped to multiple functions, and similarly, a single performance requirement may be mapped to multiple parameters.

**Function-to-Solution:** This is a mapping from function(s) to solution(s). The following cases can occur:

- A single function, $\varphi_1$, is satisfied by a single solution, $\sigma_1$, which can be expressed as $\varphi_1 \mapsto \sigma_1$.
- A single function is fulfilled by a number of (equivalent) solutions, $\varphi_1 \mapsto \{\sigma_{1.1} \vee \sigma_{1.2} \vee ... \vee \sigma_{1.n}\}$, which is termed redundancy.
- A set of components, $\sigma_{1.1}$, $\sigma_{1.2}$, ..., $\sigma_{1.n}$ collectively satisfy a single function $\varphi_1$, which can be represented as $\varphi_1 \mapsto \{\sigma_{1.1} \wedge \sigma_{1.2} \wedge ... \wedge \sigma_{1.n}\}$. That is, function $\varphi_1$ will not be fulfilled if any of these components are missing.

**Solution-to-Function:** This is a mapping from a solution (component) to a function. It is important to note that mapping relations are bidirectional, i.e. an allocation relation from function-to-solution automatically implies a relation from solution-to-function. However, unlike the Function-to-Solution mapping, where only a function satisfaction relationship is possible, here two types of relations are identified:

- Function derivation – The emergence of a new (derived) requirement/function. For example, choosing a bootstrap refrigeration system will require the air to be pressurized, which can be stated as $\sigma_1 \Rightarrow \varphi_{1.1}$.
- Function satisfaction – Assigning additional function(s) to an existing solution. In some cases, the architect may wish to allocate an existing solution to another function in order to increase performance. For instance, the utilization of the solution "jet fuel" to fulfill an extra function "Absorb heat", in addition to its primary function "provide propulsive energy". This mapping relation for multifunctional solutions can be expressed as $\sigma_i \mapsto \{\varphi_{j.1} \wedge \varphi_{j.2} \wedge ... \wedge \varphi_{j.k}\}$.

**Solution-to-model:** This is a mapping relation from solution to computational model, i.e. $\sigma_1 \mapsto \mu_1$. It specifies which behavior and performance models are associated with a particular solution. It is to be noted that in general, a single solution may be mapped to more than one model, i.e., there may be a computational model for each behaviour or performance parameter of a solution.

**Decomposition:** The following four types of decomposition relations may appear.

- Requirement decomposition – this is the breakdown of top-level requirements into sub-system and component level requirements, i.e. $\rho_1 \leftarrowtail \{\rho_{1.1} \wedge \rho_{1.2} \wedge \rho_{1.3}\}$ where the symbol $\leftarrowtail$ represents decomposition.
- Iterative function-solution decomposition – this is akin to the "zig-zagging" process mentioned earlier. It can be represented as $\varphi_1 \mapsto \{\sigma_1 \Rightarrow [\varphi_{1.i} ... \mapsto (...\sigma_{1.j})]...\}$ and comprises a series of function-to-solution and solution-to-function mappings.
- (Invariant) functional decomposition – a function $\varphi_1$ can be decomposed into a number of independent sub-functions, for example, $\varphi_{1.1}$, $\varphi_{1.2}$, and $\varphi_{1.3}$, which are not derived functions [42]. This can be written as $\varphi_1 \leftarrowtail \{\varphi_{1.1} \wedge \varphi_{1.2} \wedge \varphi_{1.3}\}$. Such decomposition may be representative of a 'functional flow', i.e. $\varphi_{1.1} \rightarrow \varphi_{1.2} \rightarrow \varphi_{1.3}$. Functional flow implies a clear process sequence, which does not need to be directly derived from the solutions, for example, source air, clean air, cool air, etc.

- Leaf function – is a function which is not refined further prior to assigning the solution.
- Logical decomposition – a solution (e.g. sub-system) can be decomposed into a number of solutions (components). This can be written as $\sigma_1 \leftarrowtail \{\sigma_{1.1} \wedge \sigma_{1.2} \wedge \sigma_{1.3}\}$

**Aggregation:** Aggregation is needed when (parts of) the functional or logical hierarchical structures are constructed from bottom up. It can also be part of a zig-zagging process, or can be done independently in a single domain. These cases would occur when, either an alternative to a reference architecture is being developed, or when only the logical/physical description is available. In the latter case the functional description needs to be "reverse engineered" as part of rationale capture. Decomposition and aggregation are applicable to both the functional and logical domains. However, during top-down design, decomposition relations are employed predominantly in the functional domain while aggregation relations are employed predominantly in the logical domain. The following concepts relate to the idea of aggregation:

- Component aggregation – is the process of combining components to comprise a single higher level component (e.g., an assembly of these components), i.e. $\{\sigma_{1.1} \wedge \sigma_{1.2} \wedge \ldots \wedge \sigma_{1.n}\} \rightarrowtail \sigma_1$. This may be the case when an integrated solution (e.g. motor gear group) replaces previously individually-linked components.
- Functional aggregation – can be the consequence of solutions-aggregation in the logical domain.

All the constructs specified above are seen as essential for enabling the functional-logical domain mapping and the capturing of the designer's actions during architecting in these domains. Ultimately, this is intended to aid not only the documentation of the final architecture, but also the process of its creation, that is, its rationale capture.

## 4.3 | Graph-Theoretic Support Structures

Presented in this section is a graph-theoretic support for the functional reasoning within and between the RFL(C) domains. Graph (data) structures are constructed by using the architectural elements and their basic relations. These data structures serve three purposes. First, they allow changes made in one domain to be recorded and traced in the other domains. Second, they assist in defining new or variant architectures. Third, these data structures can also be used as efficient means for storing architectural information (i.e., the functions, the components, and the connections between them).

**Decomposition in Requirements, Functional and Logical Domains:** Trees are employed to model the decomposition (i.e. hierarchical representation) in the requirements, functional, and logical domains. A tree, $T$, is a directed acyclic graph which has no loops and circuits. In addition, there is exactly one root node and every child has maximum one parent, i.e., there is exactly one link between any two nodes. $T$ is constructed by using only the decomposition relations, i.e. $e_i \leftarrowtail \{e_{i.1} \wedge e_{i.2} \wedge e_{i.3} \wedge \ldots \wedge e_{i.n}\}$, where $e_i$ represents an element which is decomposed into $n$ sub-elements. Each link in $T$ represents a parent-child relationship between the two connected nodes. The elements $e$ in $T$ may be a requirement, function, or solution. However, $T$ contains elements from a single domain only.

**Functional Flow, Logical Flow, and Computational Views:** Directed graphs are employed to model the functional flow, logical flow, and computational views. A directed graph (or digraph), represented by $DG(V, E)$, is a graph with a set of vertices $V$ connected by edges $E$ where each edge has a direction. A directed graph $DG$ captures the flow relations between elements of the same type. A graph, $DG_F$ proposed for functional flow modelling, describes the functions and the flow connections between them (e.g. $\varphi_1 \rightarrow \varphi_2 \rightarrow \varphi_3$). Similarly, a graph, $DG_L$, describes flows between components and subsystems in terms of energy, material or signal. $DG_F$ and $DG_L$ can be expressed in a matrix form, known as adjacency, connection, or design structure matrices. The associated adjacency matrices of $DG_F$ and $DG_L$ are denoted by $AM_F$ and $AM_L$, respectively.

**Functional-Logical Zig-Zagging:** Functional reasoning employs two types of relations between functional and

logical domains: 1) function-to-solution mapping and 2) solution-to-function mapping (also called derived functions). Here, a directed bipartite graph is used to model the functional reasoning (functional-logical zigzagging). A bipartite graph $BG$ is a graph with two disjoint sets of vertices $V_1$ and $V_2$ and a set of edges $E$ where each edge connects vertices from opposite sets. The functional reasoning directed bipartite graph, $BG$, can be expressed by Equation (1). Here, $\boldsymbol{\varphi}$ is a set of functions, $\boldsymbol{\sigma}$ is a set of solutions and $\boldsymbol{E}$ is a set of edges representing the function-to-solution and solution-to-function mappings (derived-function).

$$BG = G(\boldsymbol{\varphi}, \boldsymbol{\sigma}, \boldsymbol{E}) \tag{1}$$

Bipartite graphs are proposed to model the functional reasoning for both the complete database and the individual architectures. The $BG$ for the individual architectures (represented by $BG_A$) is the subset of the complete database $BG_D$. Figure 3 shows the bipartite graph of the complete database ($BG_D$) on the left, and the two bipartite graphs of the individual architectures ($BG_{A_1}$ and $BG_{A_2}$) on the right. It can be observed from $BG_D$ that there are two options to fulfill function $\varphi_1$. The first option is a combination of solutions ($\sigma_1 \wedge \sigma_2$), whereas, the second option is fulfilled by a single solution, $\sigma_3$ but has two derived functions, $\varphi_2$ and $\varphi_3$.

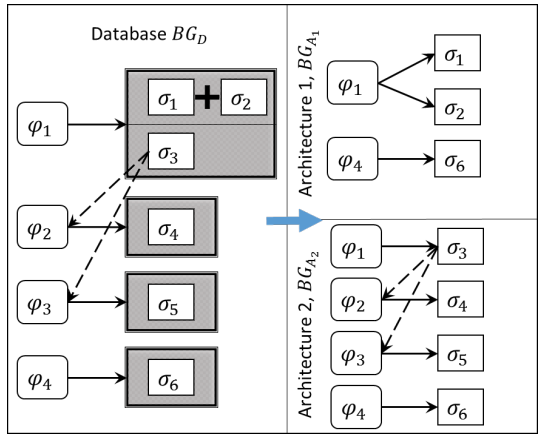


**FIGURE 3** Functional reasoning bipartite graphs

The associated matrix representations of the graphs are also useful constructs supporting the functional reasoning process. For example, the matrix representation of the bipartite graph of the complete database $BG_D$, also known as morphological matrix, contains all the leaf functions and all the available solutions for a particular project. Similarly, the matrix form of bipartite graph of the individual architectures $BG_A$, referred to as adjacency matrix or design matrix ($DM$) can be used to represent which solutions fulfill which functions (i.e. function-to-solution mappings). A similar matrix can be employed to model the solution-to-function mapping relations.

**Inter-Domain Relations:** During the systems architecting process, the architects specify mapping relations (as discussed in Section 4.2) between elements of different domains. The relations between the elements from two different domains are referred to as inter-domain relations. Requirement-to-function and function-to-solution mappings are directly specified by the architect and are referred to as direct inter-domain relations. However, these two direct relations also imply an indirect mapping relation between the requirement and the solution. These indirect relations need to be identified and managed so that the architect is able to take an informed decision while modifying the

architecture. For instance, if one or more performance requirements cannot be satisfied by a given architecture, the architect would like to determine which solutions (components) affect those performance requirements.

In the proposed framework, undirected graphs $UG$ are employed to model the inter-domain relations $\mathfrak{R}$. The $UG$ is constructed by using the direct inter-domain relations (e.g. requirement-to-function mapping, $\rho \mapsto \varphi$, function-to-solution mapping, $\varphi \mapsto \sigma$, solution-to-function mapping (derived function), $\sigma \Rightarrow \varphi$, requirement-to-parameter mapping, $\rho \mapsto p$, etc.) The elements of the R, F, L, and C domains are represented by sets, $\boldsymbol{\rho}$, $\boldsymbol{\varphi}$, $\boldsymbol{\sigma}$, and $\boldsymbol{\mu}$, respectively. The relations between the elements of any two sets, $A$ and $B$ can be represented by Equation (2), where, $A$ and $B$ represent $\boldsymbol{\rho}$, $\boldsymbol{\varphi}$, $\boldsymbol{\sigma}$, or $\boldsymbol{\mu}$.

$$A\mathfrak{R}B = \{(a, b) \,|\, a \mapsto b \wedge (a \in A \wedge b \in B) \wedge (A \neq B)\} \tag{2}$$

There are six types of possible inter-domain relations possible between the elements of the sets, $\boldsymbol{\rho}$, $\boldsymbol{\varphi}$, $\boldsymbol{\sigma}$, and $\boldsymbol{\mu}$: $\rho\mathfrak{R}\varphi$, $\varphi\mathfrak{R}\sigma$, $\sigma\mathfrak{R}\mu$, $\rho\mathfrak{R}\sigma$, $\varphi\mathfrak{R}\mu$, and $\rho\mathfrak{R}\mu$. For instance, $\rho\mathfrak{R}\varphi$ is a set of relations between a requirement and a function. Figure 4 illustrates the possible inter-domain relations, where the architect specified (direct) relations are shown by solid lines, and the indirect relations are shown by dotted lines. Algorithms for tracing the direct and indirect relations within and between the RFLC domains are presented in the next section.

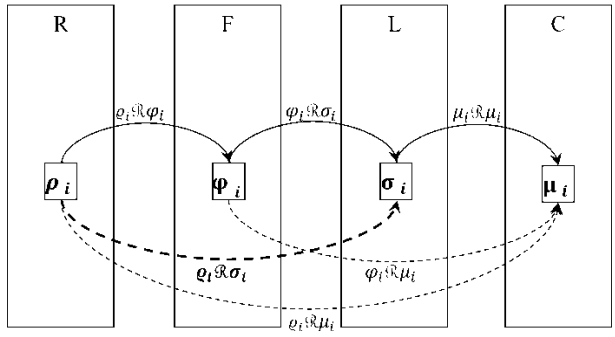


**FIGURE 4** Relations among elements of RFLC domains

## 4.4 | Tracing Algorithms

The three algorithms presented in this section are intended to work with the graph structures described in Section 4.3. The first one enables the tracing of functional-logical zigzagging, the second one allows inter-domain tracing, and the third one supports dependency tracing specifically within the computational domain. For all the algorithms, the time complexity is $O(n^2)$.

**Algorithm 1:** In order to trace the dependency between functions and solutions of a given function reasoning model, a traversal algorithm based on Depth First Search (DFS) [43] has been employed. It enables the architects to identify the affected functions and solutions by exploring the functional-logical (zigzagging) structure. If the architect wishes to modify the architecture by adding, removing or substituting a function or solution, the framework should assist by highlighting the impact of the desired change. This is illustrated in Figure 5 where the bipartite graph for the complete database is shown in the middle, and the logical and functional flow views are on the left and right, respectively. If the architect wishes to see the implication of removing solution $\sigma_4$ from the logical flow view, the

corresponding dependency (traversal) can be obtained by employing Algorithm 1, as highlighted by the red arrows. This allows to identify the affected functions, which are highlighted in grey. Function 2 ($\varphi_2$) has direct allocation relation with solution 4 ($\sigma_4$), whereas function 1 ($\varphi_1$) has indirect relation through solution 3 ($\sigma_3$).

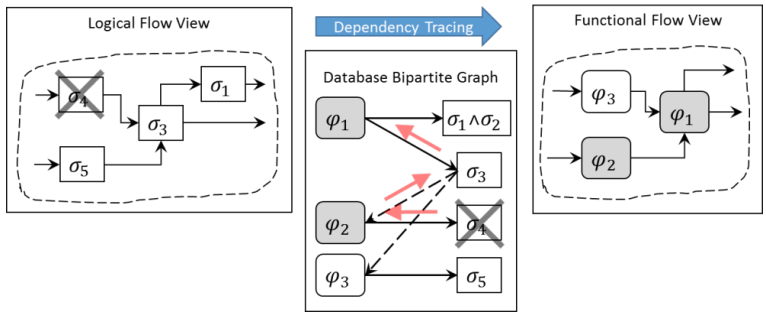


**FIGURE 5** Dependency tracing between functional and logical domains using bipartite graph

**Algorithm .1:** Tracing dependency between functional ($F$) and logical ($L$) domains

**Input** : A directed bipartite graph ($BG_A$) and architectural element ($e$) for which dependency has to be traced.

**Output**: A list $L$ of affected functions and solutions.

**Local** : Variables, $u$ and $w$ represent architectural elements that could be either function or solution.

```
Create two empty lists L and S
Create an empty dictionary V to store the status (visited or not) of elements
Initialize status of all elements of V as false (i.e., not visited)
Add e in S
while S is not empty do
    Remove last element u in S
    if status of u is false in V then
        Change status of u as true in V
    end
    Add u in L
    foreach unvisited adjacent node w of u in graph BG_A (by traversing in opposite direction) do
        Add w in S
    end
end
Return list L
```

**Algorithm 2:** The transitive closure algorithm is employed to identify all the indirect relations between elements of different domains of the systems architecture. Consider an initial graph, $G^i = (V, E)$, where, $V$ is the vertex set and $E$ is the edge set. The transitive closure graph of $G^i$ is a graph, $G^{TC} = (V, E^+)$, such that for all vertices pairs $(v_i, v_j)$ in $V$, there is an edge $(v_i, v_j)$ in $E^+$, if and only if there is a path from $v_i$ to $v_j$ in $G$. In the proposed framework, the initial graph $G^i$ is constructed with the existing direct relations only. In practice $G^i$ will be automatically generated 'behind the scene'. After applying the transitive closure on $G^i$, the resulting graph $G^{TC}$ contains both direct and indirect relations. Algorithm 2 takes architecture ($A$), and the architecture element ($e$) whose dependency is to be traced, and produces

a list of dependent elements in the other domains. Here, $e$ could be either requirement, function, solution, model or parameter. The algorithm is divided into three steps: 1) creation of an initial graph $G^i$ which stores the architect-defined (direct) relations, 2) creation of a transitive closure graph $G^{TC}$ which is based on Warshall's algorithm [44], and 3) tracing dependency of an element from a particular domain on the elements from the other domains.

**Algorithm .2:** Tracing dependency across the domains

**Input :** System Architecture, $A$, i.e. elements ($\rho$, $\varphi$, $\sigma$, and $\mu$) of $R$, $F$, $L$, and $C$ domains, and an element, $e$ whose relations with elements of other domains are to be traced.

**Output:** List $L$ of elements having (direct or indirect) relations with element $e$

**Local :** Variables $x$ and $e$ represent elements that could be either requirement, function, solution, model or parameter. Variable $c(i,j)$ represents a cell at the $i^{th}$ row and the $j^{th}$ column of a matrix.

```
Create a list E containing all elements of ρ, φ, σ, and μ
Set m equal to |E|
Create an empty matrix G^i_{m×m} storing direct relations
Assign diagonal elements of matrix G^i_{m×m} to 1
foreach i^th element x in list E do
    if x is associated to any other j^th element x' in list E then
        Set the cell at i^th row and j^th column of matrix G^i_{m×m} equal to 1
    end
end
Create a new matrix G^TC and assign G^i_{m×m} to it
foreach cell c(i,j) in matrix G^TC do
    if value at cell c is equal to 1 then
        Perform element-wise 'OR' operation on i^th row of G^TC and j^th row of G^TC
        Assign the resulting vector V_1m to the i^th row of G^TC
    end
end
Create an empty list L
foreach i^th element x in list E do
    if x is equal to e then
        foreach j^th column in matrix G^TC do
            if cell c(i,j) of matrix G^TC is equal to 1 and i is not equal to j then
                Add the j^th element of list E in list L
            end
        end
    end
end
Return list L
```

**Algorithm 3:** This algorithm supports the tracing of parameters within the computational domain. It obtains the dependency of a given parameter on other parameters, using the corresponding computational workflow. Two types of parameters are utilised from the computational domain: components' direct parameters, and components' ports parameters. The output of Algorithm 3 is an ordered list of components' direct parameters as per the sensitivity to the required parameter. An efficient method for sensitivity analysis [45] is employed to identify the most influential

direct parameters.

**Algorithm .3:** Tracing dependency within computational domain.

**Input :** Directed bipartite graph $W$ (with two types of nodes, i.e. models and parameters) representing the computational workflow; parameter $p$ whose relations with other parameters is to be traced.

**Output:** List $L$ of parameters having dependency on parameter $p$.

**Local :** Variables, $u$ and $w$ represent architectural elements that could be either model or parameter, whereas variable $p$ represents the parameter.

```
Create empty lists L, S, and K
Create an empty dictionary V to store the status (visited or not) of parameters
Initialize all elements of V as false (i.e. not visited)
Add p in list S
while S is not empty do
    Remove last element u from list S
    if status of u is false in V then
        Change status of u as true in V
    end
    Add u in list L
    foreach unvisited adjacent node w of u in graph W (by traversing in opposite direction) do
        Add w in S
    end
end
foreach parameter x in L do
    if x is not a component parameter then
        Add x to K
    end
end
foreach parameter x in K do
    Remove x from L
end
Order the parameters of L as per their sensitivity to parameter p
Return ordered list L
```

## 5 | ILLUSTRATIVE EXAMPLE

The proposed framework was demonstrated via the prototype tool "AirCADia Architect" to practicing systems architects and designers from the industrial partners in the TOICA project. These experienced engineers were acting as the “internal customers” for systems engineering from the outset. The aim of the demonstration was to obtain general feedback rather than to follow a strict evaluation and validation methodology.

During the hands-on sessions, the architects utilized both decomposition and aggregation reasoning approaches for architecture synthesis. A snapshot of the hierarchical view of the top-level requirements is shown in Figure 6.

The top-level invariant function "provide conditioned fresh air to passengers" ($\varphi_1$) was decomposed into four invariant sub-functions, i.e. "source air" ($\varphi_{1.1}$), "force airflow" ($\varphi_{1.2}$), "cool air" ($\varphi_{1.3}$) and "evacuate air" ($\varphi_{1.4}$), as shown

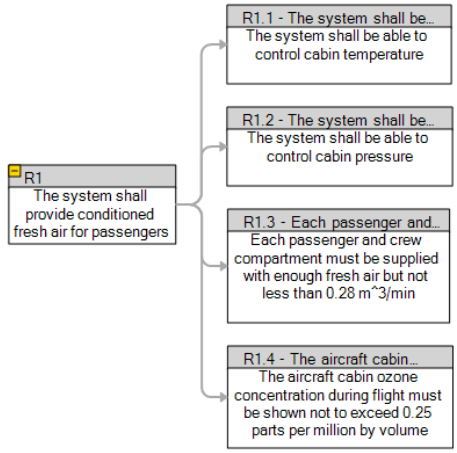


**FIGURE 6** Requirements hierarchical decomposition

in Equation (3):

$$\varphi_1 \longleftarrow \{\varphi_{1.1} \wedge \varphi_{1.2} \wedge \varphi_{1.3} \wedge \varphi_{1.4}\} \tag{3}$$

Next, solutions $\sigma_{1.i}$ were allocated to all four sub-functions $\varphi_{1.i}, i = 1, 2, 3, 4$. Two solutions were identified for the first function ($\varphi_{1.1}$): pneumatic system ($\sigma_{1.1.1}$), and ram air inlet ($\sigma_{1.1.2}$), as stated in Equation (4). It is important to note that the symbol $\vee$ (logical OR) is used in Equation (4) because only a single solution, either $\sigma_{1.1.1}$ or $\sigma_{1.1.2}$ will be allocated to $\varphi_{1.1}$.

$$\varphi_{1.1} \mapsto \{\sigma_{1.1.1} \vee \sigma_{1.1.2}\} \tag{4}$$

The second function "force air" ($\varphi_{1.2}$) was mapped to "electric fan" ($\sigma_{1.2}$), i.e. $\varphi_{1.2} \mapsto \sigma_{1.2}$. Next, two solutions were identified for $\varphi_{1.3}$: bootstrap air cycle machine ($\sigma_{1.3.1}$), and vapour cycle machine ($\sigma_{1.3.2}$), as stated in Equation (5).

$$\varphi_{1.3} \mapsto \{\sigma_{1.3.1} \vee \sigma_{1.3.2}\} \tag{5}$$

Finally, the fourth function "evacuate air" ($\varphi_{1.4}$) was mapped to $\sigma_{1.4}$ (outboard flow valve), i.e. $\varphi_{1.4} \mapsto \sigma_{1.4}$. Note that the equations presented in this section are listed only for illustration of the basic architecture description language presented in Section 4. The actual architectural synthesis in AirCADia Architect involves only clicking, dragging, and dropping actions performed by the user. After mapping solutions, $\sigma_{1.i}$, to functions, $\varphi_{1.i}$, the architects identified and created the derived functions emerging from the chosen solutions $\sigma_{1.i}$. That is, if solution "pneumatic system" ($\sigma_{1.1.1}$) is utilized for function "source air" ($\varphi_{1.1}$), then two new derived functions, "convert ozone" ($\varphi_{1.5}$) and "demist air" ($\varphi_{1.6}$) are required. Alternatively, if solution "ram air inlet" ($\sigma_{1.1.2}$) is utilized, then (in addition to $\varphi_{1.5}$ and $\varphi_{1.6}$) an extra function "pressurize air" ($\varphi_{1.7}$) is required, since unlike the air from the pneumatic system, which is already

compressed, the ram air is not. The derived functions for solutions, $\sigma_{1.1.1}$ and $\sigma_{1.1.2}$ are expressed in Equation (6).

$$\sigma_{1.1.1} \Rightarrow \{\varphi_{1.5}, \varphi_{1.6}\}, \sigma_{1.1.2} \Rightarrow \{\varphi_{1.5}, \varphi_{1.6}, \varphi_{1.7}\} \tag{6}$$

After identifying the derived functions, the architects allocated solutions to these functions. Solutions "ozone converter" ($\sigma_{1.5}$), "water separator" ($\sigma_{1.6}$) and "compressor" ($\sigma_{1.7}$) were allocated to functions, $\varphi_{1.5}$, $\varphi_{1.6}$, and $\varphi_{1.7}$, respectively. Solution "compressor" ($\sigma_{1.7}$) leads to another derived function "provide rotational energy" ($\varphi_{1.8}$), i.e. $\sigma_{1.7} \Rightarrow \varphi_{1.8}$, which was mapped to the solution "electric motor" ($\sigma_{1.8}$), i.e. $\varphi_{1.8} \mapsto \sigma_{1.8}$. It should be noted that in this case the compressor is a leaf node, because the electric motor does not 'deal' directly with the air. Since functions "pressurize air" ($\varphi_{1.7}$) and "provide rotational energy" ($\varphi_{1.8}$) were derived from solution "ram air inlet" ($\sigma_{1.1.2}$), therefore solutions, "compressor" ($\sigma_{1.7}$) and "motor" ($\sigma_{1.8}$), are relevant, as long as solution $\sigma_{1.1.2}$ is present. That is, if the architect decides to delete $\sigma_{1.1.2}$, then the derived functions $\varphi_{1.7}$ and $\varphi_{1.8}$ (and their mapped solutions $\sigma_{1.7}$ and $\sigma_{1.8}$) will automatically be deleted with the help of the elementary relations and data structures maintained in the object model. The completed functional hierarchical decomposition of the ECS is shown in Figure 7.

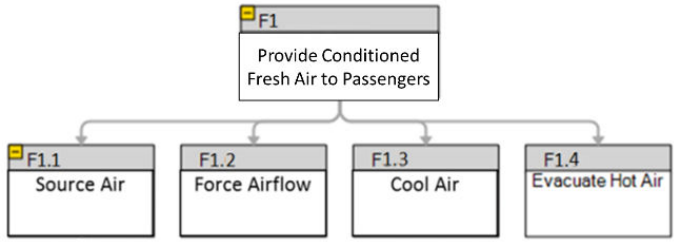


**FIGURE 7** Functional hierarchical decomposition view

The functional-logical zig-zagging is shown in Figure 8, where solutions "pneumatic system" ($\sigma_{1.1.1}$) and "bootstrap air cycle machine" ($\sigma_{1.3.1}$) are used to fulfill functions "source air" ($\varphi_{1.1}$) and "cool air" ($\varphi_{1.3}$), respectively. Here, green lines show the derived functions of the selected solution.

After the functional hierarchical decomposition and functional-logical zig-zagging views were constructed, the next step was to construct the functional and logical flow views. As shown in Figure 8, the functional reasoning model has two functions, i.e. "source air" ($\varphi_{1.1}$) and cool air ($\varphi_{1.3}$), which have more than one option to fulfill. In this case, the architects selected "pneumatic system" ($\sigma_{1.1.1}$) to fulfill "source air" ($\varphi_{1.1}$), and "bootstrap air cycle machine" ($\sigma_{1.3.1}$) for "cool air" ($\varphi_{1.3}$). Once all the relevant functions and solutions are identified, the second step is to create the connections by linking functions and solutions through their ports. AirCADia Architect enables the designers to work either in the "functional flow view" or in the "logical flow view" and, while working in one of these, the other is updated automatically to maintain consistency of information across both domains. The resulting functional and logical flow views of the conventional (bleed air) ECS architecture are shown in Figure 9.

In order to increase the reliability of the ECS, the architects decided to opt for two redundant solutions "air cycle machine packs" ($\sigma_{1.3.1}$) for function "cool air" ($\varphi_{1.3}$). The mapping of the function $\varphi_{1.3}$ with two redundant solutions is represented by Equation (7).

$$\varphi_{1.3} \mapsto \{\sigma_{1.3.1} \wedge \sigma_{1.3.1}\} \tag{7}$$

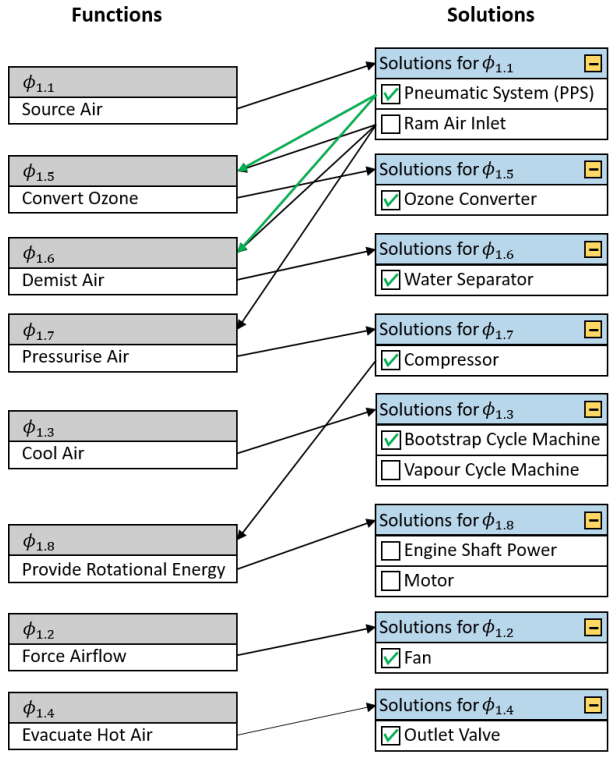


**FIGURE 8** Functional-logical zigzagging view

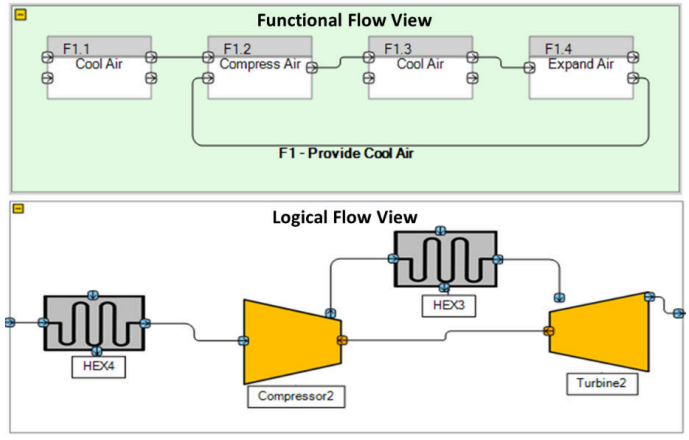


**FIGURE 9** Functional and logical flow views of the baseline ECS architecture

A hypothetical scenario was used to demonstrate the capabilities of the proposed framework regarding architecture modification. Suppose the designer wished to reduce the block fuel of the baseline architecture. Then the first

task would be to identify which of the existing components should be modified or replaced. Algorithm 3, together with the computational workflow is used to that purpose. Figure 10(a) shows the computational view, i.e. the graph of parameters (represented by ovals) and computational models (represented by rectangles). The result of the application of Algorithm 3 on this computational view is shown in Figure 10(b). It was identified that (in addition to major component options such as improving engine's specific fuel consumption and/or wing's lift-to-drag ratio) the power offtake (bleed air) of the ECS should be reduced if possible. The solid arrows in Figure 10(b) represent the dependent parameters related to the ECS.

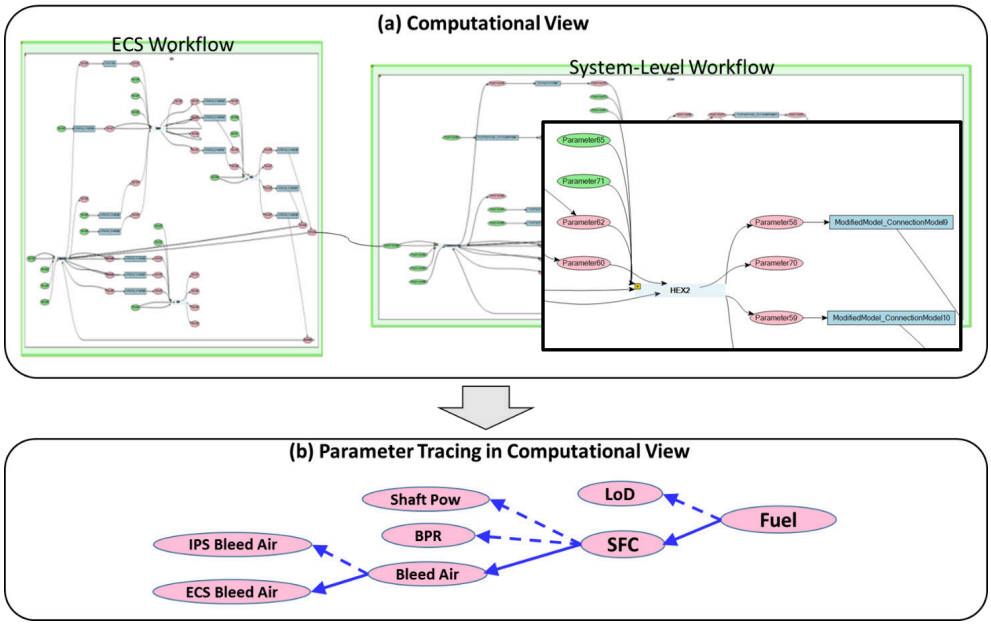


**FIGURE 10** Application of Algorithm 3: dependencies between subsystem-level and system-level parameters

Once the influential parameters are identified by Algorithm 3, the next step is to identify the affected functions and solutions by using the Algorithm 1. Suppose the designer decided to replace solution "pneumatic system" ($\sigma_{1.1.1}$) with "ram air" ($\sigma_{1.1.2}$) for function "source air" ($\varphi_{1.1}$). The rationale is that the pneumatic power is inefficient because a heat exchanger (pre-cooler) is used to cool the over compressed and overheated bleed air, discharging excess energy back into the atmosphere as waste heat. (The amount of wasted energy can reach up to 30% depending on the operating flight conditions [30]). It was identified that functions, "pressurize air" ($\varphi_{1.7}$) and "provide rotational energy" ($\varphi_{1.8}$), are the derived functions of solution, "ram air inlet" ($\sigma_{1.1.2}$), therefore they together with their allocated solutions, "compressor" ($\sigma_{1.7}$) and "motor" ($\sigma_{1.8}$) need to be added into the functional and logical flow views.

The modification of the logical flow view of the conventional (bleed) ECS to electric (bleed-less) ECS architecture is shown in Figure 11. The green dashed lines in the logical flow view enclose the added components to the baseline ECS architecture, whereas the hatched (in red) components represent the deletions.

The architects' feedback indicated that, on the whole, the approach enables the architects to effectively express their creative ideas when synthesizing architectures, and was especially acknowledged as the way forward for rationale capture [4].

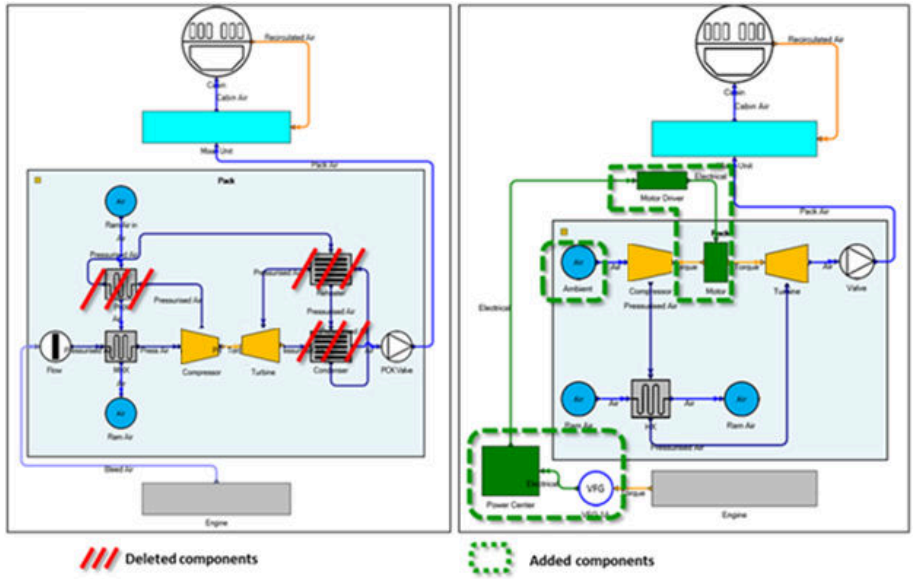


**FIGURE 11** Modification of logical flow view of the Environmental Control Systems (ECS)

## 6 | SUMMARY AND CONCLUSIONS

It was found from interviews with (airframe) systems architects that practicing engineers still prefer to work within the logical view of the architecture, rather than start with (abstract) functional decomposition as prescribed by existing systems engineering standards. Academic experiments reported in the literature also seem to support the idea that more innovative solutions could be achieved under the function-follows-forms approach. Thus while adopting the RFLP notion and the standards endorsed sequential-iterative systems engineering process, we concluded that we ought to research methods and tools, which when integrated into an innovative framework would enable the systems architects to work interactively and simultaneously in all domains without enforcing a sequence. This, however, requires that the underlying computer aided architecting system maintains traceability of the architect's choices in the RFLP domains during the process.

To this purpose, a framework specification of the mapping between the R-F-L domains was proposed which can capture their co-evolution as follows:

- The essential architecting operations within a domain and between domains (e.g., assigning/reassigning a solution to a function, derivation, decomposition, aggregation, etc.), are stated explicitly,
- The basic entities (e.g., function, requirement, port, etc.) and their relationships deemed necessary for the generic specification of the framework are defined in an object-oriented model,
- The evolving functional and logical hierarchies (trees) were modelled with the employment of existing graph-theoretic concepts such as bipartite graphs, trees, etc., and where appropriate, with (corresponding) matrix representations, referred to as design structure matrices. A number of graph-theoretic algorithms were adapted, which operate on these graph constructs and underpin the basic function reasoning operations, for example, tracing dependencies across the RFL domains, following a change (action) in one of these domains.

While adapting and incorporating some existing methods, the integration of all of the above results in a novel

framework, which enables the rapid and interactive synthesis of multiple complex systems architectures. The framework explicitly captures the 'zigzagging' pattern of the functional reasoning process, including not only allocated functions, but also the derived functions. Distinguishing between invariant and derived functions enables the architect to rapidly synthesize new architectures in the (prevalent) cases where an existing baseline is modified to this purpose. Furthermore, if the architect intends to modify a particular solution, he/she would be able to identify which of the new (derived) functions must be included, and which of the existing elements and relations of the architecture will be modified/deleted, accordingly. The traceability algorithms presented in this paper enable to identify such indirect relations through transitive closure.

A prototype tool, AirCADia Architect, was developed within the TOICA project [2] [3] for the evaluation of the framework by practicing architects. The feedback from demonstration and hands-on sessions confirmed that, on the whole, the framework enables the architects to create and manipulate architectures, and to effectively express their ideas. The proposed approach was especially acknowledged as the way forward for rationale capture.

Future work will aim to extend the systems architecting framework to the (3D) physical domain, including the development of fast methods for geometry representation and spatial layout synthesis. Work is already underway to incorporate conditional (time-dependent) views of the evolving design and to enable the synthesized architectures to be exported to widely-used computational analysis and product definition tools.

## Acknowledgements

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2013-2016, TOICA project), under grant agreement no. 604981.

## references

[1] Scholtz D. Aircraft Systems–Reliability, mass Power and Costs. In: European Workshop on Aircraft Design Education; 2002. .

[2] Rouvreau S, Mangeant F, Arbez P. TOICA—Innovations in Aircraft Architecture Selection, Uncertainty Management in Collaborative Trade-Offs. In: 6th IC-EpsMsO—6th International Conference on Experiments/Process/System Modeling/Simulation/Optimization; 2015. .

[3] Arbez P. Innovations in Aircraft Architecture Trade-Off. Proceedings of the Seventh European Aeronautics Days 2015;p. 495–500.

[4] Guenov M, Molina-Cristóbal A, Voloshin V, Riaz A, van Heerden AS, Sharma S, et al. Aircraft systems architecting a functional-logical domain perspective. In: 16th AIAA Aviation Technology, Integration, and Operations Conference Washington, D.C.; 2016. p. 3143.

[5] Bile Y, Riaz A, Guenov MD, Molina-Cristobal A. Towards Automating the Sizing Process in Conceptual (Airframe) Systems Architecting. In: 2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference; 2018. p. 1067.

[6] Crawley E, Cameron B, Selva D. System architecture: strategy and product development for complex systems. Prentice Hall Press; 2015.

[7] Maier MW. Developments in System Architecting. In: Engineering of Complex Computer Systems, 1996. Proceedings., Second IEEE International Conference on IEEE; 1996. p. 139–142.

[8] IBM, IBM Rational ROORS;. `https://www.ibm.com/us-en/marketplace/rational-doors`.

[9] Crisp H. INCOSE systems engineering vision 2020. INCOSE-TP-2004-004-02, September; 2007.

[10] Hall AD. A methodology for systems engineering. van Nostrand; 1962.

[11] Wymore AW. Model-based systems engineering. CRC press; 1993.

[12] Micouin P. Model Based Systems Engineering: Fundamentals and Methods. John Wiley & Sons; 2014.

[13] Luzeaux D. A formal foundation of systems engineering. In: Complex Systems Design & Management Springer; 2015.p. 133–148.

[14] Object Management Group. OMG Systems Modeling Language Version 1.5; 2017.

[15] Dori D. Object-Process Methodology: A Holistic Systems Paradigm; 2002.

[16] Long D, Scott Z. A primer for model-based systems engineering. Lulu. com; 2011.

[17] LML Steering Committee. Lifecycle Modeling Language; 2013.

[18] U S Department of Defence. The DoDAF Architecture Framework Version 2.02; 2010.

[19] ANSI/EIA 632, Processes for Engineering a System. American National Standards Institute/Electronic Industries Alliance; 1999.

[20] ISO/IEC/IEEE 15288:2015, Systems and software engineering – System life cycle processes. International Organization for Standardization Geneva; 2015.

[21] Pahl G, Beitz W. Engineering design: a systematic approach. Springer Science & Business Media; 2013.

[22] Stone RB, Wood KL. Development of a Functional Basis for Design. Journal of Mechanical design 2000;122(4):359–370.

[23] Hirtz JM, Stone RB, Szykman S, McAdams DA, Wood KL. Evolving a functional basis for engineering design. In: Proceedings of the ASME Design Engineering Technical Conference: DETC2001, Pittsburgh, PA; 2001. .

[24] Freeman P, Newell A. A Model for functional reasoning in design. In: Proceedings of the 2Nd International Joint Conference on Artificial Intelligence IJCAI'71, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 1971. p. 621–640.

[25] Chakrabarti A, Bligh TP. A scheme for functional reasoning in conceptual design. Design Studies 2001;22(6):493–517.

[26] Umeda Y, Tomiyama T, Yoshikawa H. FBS modeling: modeling scheme of function for conceptual design. In: Proceedings of the 9th international workshop on qualitative reasoning; 1995. p. 271–278.

[27] Suh NP. The Principles of Design (Oxford Series on Advanced Manufacturing) 1990;.

[28] Suh NP. Axiomatic Design: Advances and Applications (The Oxford Series on Advanced Manufacturing) 2001;.

[29] Haskins C. Systems engineering handbook: A guide for system life cycle processes and activities. International Council on Systems Engineering; 2007.

[30] VDI 2206, Design methodology for mechatronic systems. Verein Deutscher Ingenieure; 2004.

[31] Kleiner S, Kramer C. Model based design with systems engineering based on RFLP using V6. In: Smart Product Engineering: Proceedings of the 23rd CIRP Design Conference Bochum, Germany: Springer; 2013.p. 93–102.

[32] Dassault Systemes, Catia Systems Engineering;. https://www.3ds.com/products-services/catia/disciplines/systems-engineering/.

[33] Siemens, Teamcenter Systems Engineering;. https://www.plm.automation.siemens.com/global/en/products/collaboration/mbse-model-based-systems-engineering.html.

[34] Smith SM, Ward TB, Finke RA. Paradoxes, principles, and prospects for the future of creative cognition. The creative cognition approach 1995;p. 327–335.

[35] Howard TJ, Culley SJ, Dekoninck E. Describing the creative design process by the integration of engineering design and cognitive psychology literature. Design studies 2008;29(2):160–180.

[36] Sagiv L, Arieli S, Goldenberg J, Goldschmidt A. Structure and freedom in creativity: The interplay between externally imposed structure and personal cognitive style. Journal of Organizational Behavior 2010;31(8):1086–1110.

[37] Vermaas PE. On the formal impossibility of analysing subfunctions as parts of functions in design methodology. Research in Engineering Design 2013;24(1):19–32.

[38] Nickerson RS. Enhancing Creativity. In: Sternberg RJ, editor. Handbook of creativity Cambridge, UK: Cambridge University Press; 1999. .

[39] Sawyer RK. Explaining creativity: The science of human innovation. Oxford University Press; 2011.

[40] Greene MT, Gonzalez R, Papalambros PY, McGowan AM. Design Thinking Vs . Systems Thinking for Engineering Design : What's the Difference ? ICED 2017 conference proceedings 2017;.

[41] Balachandran LK, Guenov MD. Computational workflow management for conceptual design of complex systems. Journal of Aircraft 2010;47(2):699–703.

[42] Faisandier A. Systems architecture and design. Sinergy'Com; 2013.

[43] Tarjan R. Depth-first search and linear graph algorithms. SIAM journal on computing 1972;1(2):146–160.

[44] Warshall S. A theorem on boolean matrices. Journal of the ACM (JACM) 1962;9(1):11–12.

[45] Chen X, Molina-Cristóbal A, Guenov MD, Riaz A. Efficient method for variance-based sensitivity analysis. Reliability Engineering and System Safety 2019;.

**Marin D Guenov** is Professor of Engineering Design and Aerospace Engineering at Cranfield University. He has over 35 years industrial and research experience in Engineering Design and Multi-Disciplinary Optimization, gained in the materials handling, marine and aerospace sectors. Over the last fifteen years he conceived and led the development of AirCADia, a research prototype tool for model-based design. The specifications presented in this paper are related to AirCADia-Architect, a module for system synthesis. Professor Guenov was appointed as the Head of the Aerospace Engineering Department at Cranfield University in 2010 and in 2013 became the inaugural Head of the University's Centre of Excellence in Aeronautics. He is a Fellow of the Institution of Mechanical Engineers, a Fellow of the Royal Aeronautical Society, a Senior Member of the AIAA and a Chartered Engineer. Professor Guenov is the current Chairman of the Cranfield Branch of the Royal Aeronautical Society.

**Atif Riaz** is a Lecturer in Engineering Design at Cranfield University, UK. He studied BEng in Aeronautics and Astronautics from the University of Southampton, and PhD in Aerospace Engineering from the Cranfield University. His research interests focus on the development of methods and tools for complex systems design and analysis, including Multidisciplinary Design and Optimisation

(MDAO), Set-Based Design (SBD), and Model-Based Engineering. He has been a key collaborator and academic advisor to multiple research projects for the UK/EU aerospace industry, in partnership with the AIRBUS, Roll-Royce and others.

**Yogesh H Bile** is a Mechanical Engineer in New Product Development (NPD) at Schlumberger, UK. He received a B.E. in Mechanical Engineering from the Mumbai University, India and M.Tech. in Thermal Science and Engineering from Indian Institute of Technology (IIT), Kharagpur, India. After masters, he worked as a Deputy Manager in Methods Development Group (MDG) at Mahindra and Mahindra Ltd., India. Here, he was involved in development of 1D (low fidelity) simulation models for automotive and tractor systems. He received 'Mahindra Rise Award' for successfully deploying model management system resulting in improvements in knowledge retention and the efficiency of company-wide design processes. He left the company to pursue PhD in Aerospace Engineering from Cranfield University. He is a member of the Institute of Engineering and Technology (IET). His research interests include mathematical modelling of complex engineering systems and application of the artificial intelligence, data science and machine learning methods to improve efficiency and effectiveness of engineering design processes.

**Arturo Molina-Cristobal** graduated with a BSc in Mechanical Engineering with a major in Mechatronics from Universidad Iberoamericana, Mexico, in 1996. In 2005, he completed his PhD in the Multiobjective Control at the University of Sheffield. He has an established academic career with over fifteenth years of research experience at world-renowned engineering institutions such as Sheffield University, Cambridge University, Cranfield University and presently, University of Glasgow Singapore. From 2011 to July 2019, he is a Lecturer in Engineering Design at Cranfield University. He has been a key collaborator and academic advisor to multiple high-profile projects for the UK and European aerospace industry; in partnership with the AIRBUS group; such as, the EU-FP6 MOET, EU-FP7 Crescendo, UK-TSB CONGA, EU-FP7 TOICA and UK-ATI APROCONE. Currently, he is an Assistant Professor in Mechanical Engineering at the University of Glasgow Singapore. His work is focused on developing engineering methods in areas such as Multidisciplinary Design Optimisation (MDO), Uncertainty Quantification and Management (UQM), robust optimization design (RDO), Multi-objective design optimization, and Model-Based Systems Engineering (MBSE).

**Albert (Stevan) van Heerden** is a Research Fellow in Airframe Systems at Cranfield University. He obtained a BEng in Mechanical Engineering from the University of Pretoria, after which he spent four years in the aerospace industry as an aerodynamicist and control engineer. He then continued

his studies, obtaining a Master's of Science in Aeronautics from the California Institute of Technology and a PhD in Aerospace Engineering from Cranfield University. Albert's research interests include the modeling and simulation of complex engineering systems and aerospace vehicle and airframe systems design, with a particular focus on thermal management systems.

2020-02-17

# Computational framework for interactive architecting of complex systems

Guenov, Marin D.

Wiley

Guenov MD, Riaz A, Bile Y, et al., (2020) Computational framework for interactive architecting of complex systems. Systems Engineering, Volume 23, Issue 3, May 2020, pp. 350-365 https://doi.org/10.1002/sys.21531