System Architecture Design Space Exploration: Integration with Computational Environments and Efficient Optimization

Jasper H. Bussemaker*, Luca Boggero†, Björn Nagel‡
DLR (German Aerospace Center), Institute of System Architectures in Aeronautics, Hamburg, Germany

System Architecture Optimization (SAO) enables automatically exploring combinatorial system architecture design spaces, which can reduce bias and enable more architectures to be considered in early design phases. This paper presents the Architecture Design Space Graph (ADSG), a directed graph for modeling architecture design spaces, and encoding them as optimization problems to be solved by optimization algorithms. The ADSG is built on top of the Design Space Graph (DSG), which models hierarchical design spaces using selection and connection choices, where selection choices define which nodes are selected in architecture instances, and connection choices represent source-to-target connection problems. Selection and connection choice encoders are introduced that enable full enumeration of valid design vectors, and ensure any source-to-target connection problem can be encoded such that optimization algorithms can effectively search the design space. The ADSG extends the DSG for use in system architecting, by defining nodes such as functions, components and ports, which enables function-based architecture definition. The ADSG can be modeled using ADORE: a Python tool with a web-based GUI that allows connecting to performance evaluation code through Python-based and file-based interfaces, and connecting to open-source architecture optimization algorithms. The presented method is demonstrated by three application cases, each demonstrating different evaluation and optimization aspects: a multi-stage launch vehicle, a guidance, navigation and control system, and a jet engine architecture. It is shown that SAO problems formulated using ADORE perform as well, if not better, than manually-defined SAO problems, without requiring the user to be an expert in formulating optimization problems.

I. Introduction

The architecture of a system describes what components that system consists of, and how these components are combined to achieve the system goals [1]. It transforms system requirements into a blueprint for implementation, and provides an input to detailed design phases. An architecture assigns elements of form (i.e. components) to function: function specifies *what* the system should perform; form specifies *how* the architecture performs its functions. The architecture of a system greatly influences project success [2], however decisions regarding architecture are taken early in the project and are therefore subject to great uncertainty [3]. Therefore, significant modeling and design effort is required before alternative architectures can be compared to each other to select the most appropriate architecture for a given design problem. This is why often only a few architecture alternatives are considered and compared, and architecture trade-off might be subject to bias, overconfidence or conservatism [4, 5].

System Architecture Optimization (SAO) is an emerging field that applies numerical optimization algorithms to automatically search an architecture design space, thereby considering more architectures and resulting in less bias and a more complete overview of promising architecture instances [6, 7]. To implement an SAO problem, the architecture design space should be defined, it should be possible to quantitatively evaluate the performance of architecture instances, and appropriate optimization algorithms should be available [8]. This paper provides an overview of how to perform SAO using the Architecture Design Space Graph (ADSG): a directed graph that combines architecture elements (e.g. functions and components) with decision models, enabling the automatic generation of architecture candidates. The ADSG can be encoded as an optimization problem in terms of design variables x, objectives f and constraints g, which can be used by optimization algorithms to explore the design space. Compared to other methods, the presented approach thus enables architecture design space exploration without requiring the exhaustive generation of all architecture,

*Researcher, DDP Group, Department of Digital Methods for System Architecting, jasper.bussemaker@dlr.de
†Head of DDP Group, Department of Digital Methods for System Architecting, luca.boggero@dlr.de
‡Institute Director, Institute of System Architectures in Aeronautics, Hamburg, bjoern.nagel@dlr.de

leveraging modern optimization algorithms for efficient exploration, and by allowing the user to define the design space using system architecting terminology (functions, components, etc.).

This paper continues with a general overview of SAO in Section II. The ADSG is presented in more details in Section III, focusing on extensions done since the paper originally introducing the ADSG [9], in particular the underlying Design Space Graph (DSG) and the modeling environment ADORE. In Section IV we demonstrate the ADSG by three SAO problems modeled in ADORE: a multi-stage launch vehicle, a guidance, navigation and control system, and a jet engine. Section V concludes the paper.

II. System Architecture Optimization Review

On a high level, any optimization problem consists of two elements: an algorithm to suggest design solutions (the optimization algorithm) and a function to tell the algorithm how good a given solution is (the evaluation function) [10]. Similarly, an SAO problem consists of [11]:

1) An architecture generator, suggesting architecture instances to be evaluated.
2) An architecture evaluator, evaluating the performance of a given architecture instance.

A. Architecture Generator

The architecture generator is an automated version of the architecture synthesis [12] or concept generation [1, 13] step in systems engineering. It generates architecture instances from an associated architecture design space. The architecture design space determines which architectures can be generated, by modeling architectural choices and constraints. It should be specified formally enough so that automatic reasoning by a computer program is possible. However, it should be possible to specify it in terms a system engineer would be familiar with, such a function and form [14].

A system model created in a Model-Based Systems Engineering (MBSE) setting typically represents a specific architecture instance or several architecture instances [14]. To *model an architecture design space*, architectural choices should be modeled and architecture instances should be generated by resolving choices. Modeling choices in an MBSE context is made possible by specific extensions of the Systems Modeling Language (SysML) such as CVL [15] or VAMOS [16]. Variability is an integral part of SysMLv2 [17], showing that it is considered an important capability to be supported in the future. Other approaches include variability modeling using feature models [18, 19], function-means models [20], or configurable components [21].

Generating architecture instances from some architecture design space model can either be done exhaustively (architecture enumeration) or selectively (architecture optimization). Several *architecture enumeration* methods based on graph grammars [22] have been developed in the past, including ArchEx [23], perfect matchings [24], RoMOGA [25] and Automatic Topology Generation [26]. Siemens Simcenter Studio enables architecture enumeration by modeling architecture elements and port connection constraints, and then solving a constraint satisfaction problem (CSP) [27, 28]. The morphological matrix is a well-developed method for modeling system variability, and when combined with a (in)compatibility matrix it can be used to enumerate architectures, as for example implemented by ARMA [29], IRMA [30], or shown by [31]. Other methods for architecture enumeration include functional flows [32–34], resource flows [35], Architecture Decision Graph (ADG) [36], RAAM [37], and using description logic reasoners [38].

The main downside of architecture enumeration is that architecture design spaces can be extremely large: for example 79 million architecture are possible in the GN&C problem in [39], and a satellite instrument selection problem in [40] features 8.8 trillion possible architecture. This makes architecture enumeration impractical and/or infeasible due to time or computational resource constraints. In SAO, architectures are generated selectively and quantitative evaluation is used to steer the exploration towards the best architecture(s) instead. Feature models [41, 42] and the morphological matrix [6, 43, 44] have found limited application in architecture optimization. Apaza and Selva present the Architecture Decision Diagram (ADD), which enables the user to model architectural decisions [39] based on architecture decision patterns [45], which are then encoded as design variables and dedicated search and repair operators for use in evolutionary algorithms [40, 46]. These architecture optimization methods, however, do not represent architecture concepts like function and form, no links are established to system requirements, and often no continuous design variables are supported. The ADSG [9], see also Section III, aims to close this gap by combining a function-based architecture superset model (150% model) with a variability model. The ADSG can be encoded as design variables and a repair operator is made available for use by optimization algorithms. This enables the user to model the architecture design space with elements familiar to systems engineers, and at the same time leverage the strength of optimization algorithms to search this design space.

Evolutionary Algorithms (EA) have been used to solve architecture optimization problems [6, 40, 47] due to their ability to effectively search the mixed-discrete, hierarchical design space [48]. Although more practical than full enumeration of the architecture design space, evolutionary algorithms might still require in the order of thousands of evaluations and are therefore not appropriate for expensive evaluation functions, for example resulting from physics-based simulation. To solve this, recently Surrogate-Based Optimization (SBO) and in particular Bayesian Optimization (BO) algorithms have been applied to architecture optimization [48]. SBO algorithms build a surrogate model of the objective function $f(\mathbf{x})$ and constraint functions $g(\mathbf{x})$, and use this model to predict where the best architecture(s) are using infill functions. BO algorithms use surrogate models that also provide uncertainty information, such as Gaussian Process (GP) models, to increase the predictive accuracy of these infill functions. Recently, BO for architecture optimization has been extended to also support hidden constraints [49]: evaluation failures resulting in no f and g values being available, which is a common phenomena for evaluation using physics-based simulation. For more details about architecture optimization algorithms and BO in particular, the interested reader is referred to [48, 49].

B. Architecture Evaluator

The architecture evaluator represents the evaluation function in an optimization process [11]: it returns performance metrics for a given architecture instance. The evaluation function is problem-specific and can be implemented as anything ranging from a simple script using lookup tables or low-fidelity handbook equations, to distributed multi-disciplinary high-fidelity simulation toolchains, as long as it adheres to the following requirements:

- The performance metrics should be *sensitive* to all relevant architectural decisions.
- The performance metrics should be *available* for all possible architectures, and with similar accuracy/fidelity.
- The evaluation function should be executable *without user interaction* when running the optimization.

Beyond problem-specific ad-hoc code, some research has been performed on general methodologies for integrating system architecture models with system simulation, as this would improve consistency and enables verification and validation [50]. Performance calculations can be integrated in architecture models directly (e.g. [42, 51]) or external tools can be connected (e.g. [52, 53]). Especially useful for simulating system-level performance is Multidisciplinary Design Analysis and Optimization (MDAO) [50]: a set of methods and technologies for numerically coupling disciplinary analysis tools, ensuring tightly-coupled computations are consistent with each other [54].

Helle et al. [55] present a method for modeling variability-aware analysis architectures in SysML using Parametric Analysis Models (PAM), and executing analysis architectures from architecture instances manually defined from a superset model. MDAO for architecture optimization was partly supported in work by Bussemaker et al. [56], where generated architectures provided input for a collaborative MDAO workflow. Bruggeman et al. [57] present an MDAO workflow that dynamically swaps a sub-workflow depending on a selection of the manufacturing process of the part being designed. Sonneveld et al. [58] demonstrate an MDAO workflow that dynamically modifies a sub-problem based on the number of ribs selected in an aileron. The integration of these methods into a collaborative MBSE and MDAO framework is presented in [59]. Full architecture optimization with MDAO was demonstrated by Bussemaker et al., with dynamically-instantiated MDAO workflows for evaluating jet engine [60] and hybrid-electric propulsion system [61, 62] architectures. Garg et al. [63] present a method for modeling computational variability due to architecture variability directly in MDAO workflows, enabling MDAO workflows to dynamically alter their behavior based on the architecture being evaluated without requiring reformulation.

III. The Architecture Design Space Graph (ADSG)

This section presents and discusses the Architecture Design Space Graph (ADSG) and its use for SAO, at three abstraction levels:

- *Design Space Graph (DSG)*: the core mechanism for modeling hierarchical choices using a directed graph, and encoding choices as design variables for defining optimization problems.
- *Architecture Design Space Graph (ADSG)*: the application of the DSG for system architecting, defining node types like functions, components, and ports.
- *ADORE*: the editor that provides a web-based graphical user interface (GUI) for creating ADSG models, and application programming interfaces (APIs) for connecting to optimization libraries and evaluation code.

The ADSG and ADORE were first introduced in [9] and [56], respectively. Citations are omitted from the following sections for clarity.

A. The DSG: Choice Modeling and Design Problem Formulation

The Design Space Graph (DSG) implements the mechanism for modeling hierarchical architecture choices and encoding design variables, and has recently been abstracted from the original ADSG implementation presented in [9]. This abstraction provides possibilities for more focused development and the potential for reusing the same mechanism in optimization contexts other than SAO. The purpose of the DSG namely is to model the hierarchical structure between choices and nodes, and nodes can be interpreted as elements that can be part of some architecture instance. The Python implementation of the DSG is available open-source as ADSG CORE* (named as such because it represents the core mechanism of the ADSG). Following sections provide more details about how choices are modeled, how architecture instances are generated, and how choices are encoded into design variables.

1. Choice Modeling and Architecture Generation

CRAWLEY ET AL. argue that the basic tasks of a system architect involve decomposing form and function, mapping function to form, specializing and characterizing form and function, and connecting form and function [1]. SELVA ET AL. observe various reoccurring patterns in architecture decisions [45], and map above architecting tasks to these patterns. The ADSG has originally been designed assuming that the first tasks (decomposing, mapping, specializing, and characterizing) involve selecting architecture elements (function or form) to be included in an architecture instance, and that the connecting task remains as a last task and depends on which elements have been selected by prior tasks. Another assumption is that the choices involved in element selection are tightly coupled and hierarchical, with many choices only activated based on other choices. For this purpose, the DSG is a directed graph consisting of two domains:

- Generic nodes, derivation edges, incompatibility edges and selection choices for *selecting* the elements that are included in an architecture instance.
- Connector nodes, connection edges and connection choices for modeling *connection* tasks.

Selection Choices A *derivation edge* is a directed edge that asserts that if the source node is included in an architecture instance, then the target node is also included in that same instance. Generic nodes can have any number of incoming and outgoing derivation edges. Multiple outgoing derivation edges mean that all target nodes are selected if the node is selected; multiple incoming derivation edges mean the node is selected if any of the source nodes are selected. A *selection choice* node (called option-decision in [9]) represents an architectural choice where one of the mutually-exclusive option nodes is selected. When resolved, the (singular) incoming generic node is connected to the selected option node by a derivation edge. The nodes not selected and their derived nodes, excluding confirmed nodes (see below), are removed from the graph. One or more generic nodes are designated as *start* nodes: these nodes and their derived nodes (not passing through selection choice nodes) are present in all architectures and therefore are designated *permanent* nodes. Non-permanent nodes that have at least one path originating from a starting node (passing through one or more selection choice nodes) are *conditional* nodes. When resolving a DSG to an architecture instance, nodes derived from a starting node are designated *confirmed* to highlight the fact that they are part of a specific architecture instance. Choice nodes are *active* if they are or have been confirmed, *inactive* otherwise. An *incompatibility edge* is an undirected edge that asserts that if either of the two nodes is confirmed, the other node and its derived nodes are not. The DSG is infeasible if an incompatibility edge is defined between two permanent nodes. A DSG containing no more selection or connection choices is designated *final*. A final and feasible DSG represents an architecture instance.

Figure 1 shows an example DSG with 12 nodes (Nx), 2 selection choice nodes (Cx) and N1 the starting node. Table 1 lists for the same example which nodes are permanent or conditional, and for an enumeration of C1 and C2 options which nodes are confirmed or infeasible. There are 12 states for this DSG: 1 initial state (only containing permanent nodes), 3 partial states (C1 is resolved; C2 not) and 8 final states (including 2 infeasible and 6 feasible states).

Connection Choices *Connection choices* (called permutation-decisions in [9]) offer a generic way to model source to target connection problems, where source and target nodes are represented using *connection nodes*. Connection nodes behave the same as generic nodes with respect to derivation edges and selection choice, however additionally specify a *connector constraint*: a specification of how many outgoing (source) or incoming (target) connections the associated connector node can accept, and whether repeated connections to/from the same target/source node are allowed. The connector constraint can be specified as a list of numbers (e.g. 1, 2 or 3 connections: `1,2,3`), a lower and an upper bound (e.g. between 0 and 3, inclusive: `0..3`), or only a lower bound (e.g. 1 or more: `1..*`).

*https://adsg-core.readthedocs.io/

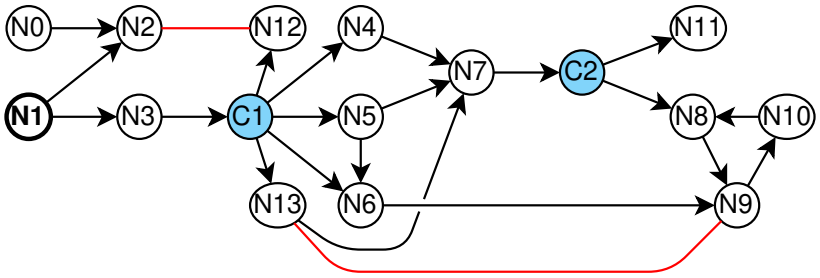


Fig. 1 DSG example with generic nodes N and selection choice nodes C (shown in blue). Edges include derivation edges (black arrows) and incompatibility edges (red). Node N1 is the start node.

Table 1 Node status table for all possible combinations of choice options for the example shown in Figure 1. In the bottom part of the table, ✓ represents confirmed nodes in the associated (partial) architecture. × represents violated incompatibility constraints. Node names in the choice columns (C1 and C2) indicate the selected option node.

Node	N0	N1	N2	N3	C1	N4	N5	N6	N7	C2	N8	N9	N10	N11	N12	N13
Permanent		✓	✓	✓	✓											
Conditional						✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Partial arch. 1		✓	✓	✓	N4	✓			✓	✓						
Architecture 1		✓	✓	✓	N4	✓			✓	N8	✓	✓	✓			
Architecture 2		✓	✓	✓	N4	✓			✓	N11				✓		
Partial arch. 2		✓	✓	✓	N5		✓	✓	✓	✓	✓	✓	✓			
Architecture 3		✓	✓	✓	N5		✓	✓	✓	N8	✓	✓	✓			
Architecture 4		✓	✓	✓	N5		✓	✓	✓	N11	✓	✓	✓	✓		
Architecture 5		✓	✓	✓	N6			✓			✓	✓	✓			
Infeasible arch. 1		✓	×	✓	N12										×	
Partial arch. 3		✓	✓	✓	N13				✓	✓						✓
Infeasible arch. 2		✓	✓	✓	N13				✓	N8	✓	×	✓			×
Architecture 6		✓	✓	✓	N13				✓	N11				✓		✓

All architecture decision patterns [45] can be represented as source to target connections and connector constraints. However, for some patterns the order in which connections are established is not relevant (for example the partitioning pattern). To model this, a *connection grouping node* can be used: the connector constraint of this node depends on aggregated connector constraints of incoming connection nodes (connected by derivation edges). Finally, it is also possible to define combinations of source and target nodes that may not be connected using *exclusion edges*. Using connection (grouping) nodes, connection edges and connection choice nodes, it is possible to represent all architecture decision patterns identified by Selva et al., as shown in Table 2. That table also contains two additional patterns: unordered (non-replacing) combining, which represents combining patterns where the order of option selection is not relevant. Due to its flexible formulation, however, also other connection choices can be modeled. For example, in [64] connection choices derived from safety regulations were modeled, specifying that each electric brake actuator should be connected to at least 2 independent electrical power sources.

Figure 2 shows an example DSG with a connection choice (C2). Its sources are the connection grouping node Grp and connection node S1; its targets are connection nodes T1 and T2. The connection edges, shown by dashed black arrows, display the connection constraints. The connection constraint of the connection grouping node Grp is aggregated from constraints by its underlying source connection nodes S1 and S2: each of these can have either 1 or 2 outgoing connections, which is aggregated to 2, 3 or 4 connections. The selection choice C1 determines whether S2 is confirmed (S1 is always confirmed). If S2 is not confirmed, only S1 remains and the aggregated connection constraint of Grp is modified to 1 or 2 connections (i.e. derived from S1 only). Table 3 enumerates all *valid connection sets* that exist for the example of Figure 2. The example demonstrates that this way of modeling connection choices can lead to non-trivial valid connection sets that extend the possibilities of architecture decision patterns as identified by Selva et al. [45].

Table 2 Architecture decision patterns modeled using connection choices. n and m are independent integers equal to or greater than 1; n @ cc specifies the number of nodes (n) with connection constraint cc; $(i,)$ represents a connection from source i to target j; "(rep)" indicates repeated connections are allowed.

Pattern	Source nodes	Target nodes	Excluded edges
Combining	n @ 1	m @ 0, 1	
Unordered combining	1 @ n (rep)	m @ 0..$*$ (rep)	
Unordered non-replacing combining	1 @ n	m @ 0, 1	
Assigning	n @ 0..$*$	m @ 0..$*$	
Partitioning	n @ 0..$*$	m @ 1	
Downselecting	1 @ 0..$*$	m @ 0, 1	
Connecting	n @ 0..$*$	n @ 0..$*$	(i, j) if $i \geq j$
Permuting	n @ 1	n @ 1	

A connection choice is resolved by applying valid connection edges (i.e. edges adhering to all connector constraints) directly from source to target nodes and removing the connection choice node. Connection choices are resolved independently of other connection choices, however they depend on selection choices. Connection choices are therefore resolved after all selection choices have been resolved, making it possible that some or all of the source and/or target nodes have been removed because they are not selected. Effectively, this means that a different connection choice is defined for each source and target node *existence scenario*. If for a given existence scenario it is not possible to establish valid connection sets adhering to all connector constraints, the associated DSG is infeasible.

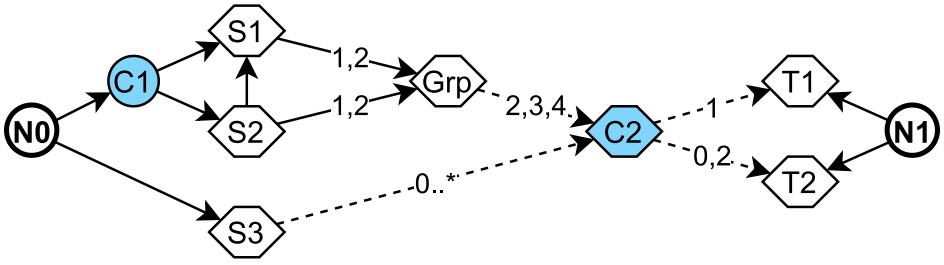


Fig. 2 DSG example with generic nodes N, choice nodes C (blue). Connection (choice) nodes are shown as hexagons; connection edges as dashed lines. Nodes N0 and N1 are the start nodes.

Table 3 Valid connection sets for the connection choice C2 shown in Figure 2.

	Connections from / to Grp		Connections from / to S3		Total connections			
S2 exists	T1	T2	T1	T2	Grp	S3	T1	T2
Yes	0	2	1	0	2	1	1	2
	1	1	0	1	2	1	1	2
	1	2	0	0	3	0	1	2
No	0	1	1	1	1	2	1	2
	1	0	0	0	1	0	1	0
	1	0	0	2	1	2	1	2
	0	2	1	0	2	1	1	2
	1	1	0	1	2	1	1	2

Design Problem Definition Next to selection and connection choices, it is also possible to define generic design variables, for example to model parameter selections. These are defined using *design variable nodes* that are subject to node selection just as generic nodes and can therefore exist conditionally.

Choice constraints allow constraining option availability for choices based on other choices. For continuous design variables only linking is possible: here the same value relative to the respective design variable bounds is applied. For discrete design variables and selection and connection choices, four types of choice constraints are available: linked, permutations, unordered combinations, and unordered non-replacing combinations. These constraints are applied by applying the following logic:

- *Linked*: all choices are assigned the same option index, e.g. AA, BB, CC.
- *Permutations*: all choices have a different option index, e.g. AB, AC, BA, BC, CA, CB.
- *Unordered combinations*: equal or higher index than preceding choices, e.g. AA, AB, AC, BB, BC, CC.
- *Unordered non-replacing combinations*: higher index than preceding choices, e.g. AB, AC, BC.

The design problem definition is completed by additionally defining performance metrics using *metric nodes*. Metric nodes represent outputs of the architecture evaluation and can be used as objectives f or constraints g in the context of a design problem. Objectives are minimization or maximization targets and metric nodes can only be used as objectives if they are permanent, as otherwise it is not possible to compare the performance of all architectures. Constraints represent inequality design constraints: values that should be above (greater than or equal) or below (lower than or equal) some threshold. Metrics used as constraints can be conditional: if the node is not part of some architecture, it means that the constraint does not apply and the constraint is assumed satisfied (the value is set equal to the threshold).

2. Design Problem Encoding, Decoding and Correction

As presented in the preceding section, a DSG defines an architecture optimization problem using selection choices, connection choices, generic design variables, various types of constraints, and metrics. Here we present how this design space definition is encoded as a set of mixed-discrete design variables x to be used by an optimization algorithm and how a given design vector $\mathbf{x}$ is decoded into an architecture instance. We additionally present how design vectors are corrected and imputed to ensure they are valid [48] (representing a feasible and unique architecture instance).

Encoding and Decoding Selection Choices Selection choices can be encoded by two encoding algorithms: the fast algorithm and the complete algorithm. The complete encoder results in more efficient design variable definitions and enables the exhaustive identification of all valid design vectors. This, however, requires significantly more computational resources (time, memory) than the fast encoder, which directly maps selection choices to discrete design variables. When encoding the design space, therefore first the complete encoder is tried. If some time or memory limit is reached, the fast encoder is used instead.

The *fast encoder* maps selection choices to discrete design variables, with option nodes mapped to integer values between 0 and n_{opts}-1 for each selection choice. For choice groups constrained by a linked choice constraint (i.e. all choices are assigned the same option index), only one discrete design variable is defined. Remaining choices are known as *forced selection choices*, because their value is fully determined by previously-taken selection choices. Decoding and correcting design vectors is performed in a greedy manner: starting from the initial DSG, confirmed (active) selection choices are assigned options from the given design vector. If for a given selection choice the requested option node is not available, the requested option node is corrected to the closest available option, thereby ensuring that the design vector represents a valid architecture.

The *complete encoder* exhaustively identifies all possible discrete design vectors $x_{\text{valid,discr}}$ and associated activeness $\delta_{\text{valid,discr}}$ and node existence information. This enables improving optimizer performance by using hierarchical sampling and correction algorithms developed in [48]. Additionally, it enables the calculation of the number of valid architectures, the identification of *forced selection choices*, and the identification of all existence scenarios as needed for encoding connection choices. Decoding and correcting design vectors is done by first ensuring that the requested design vector is valid: if this is not the case, the closest valid design vector as measured by the Manhattan distance is selected instead. Table 4 shows valid design vectors, activeness information, and node existence for all feasible architecture shown in Table 1. Selection choices C1 and C2 are mapped to design variables x_0 and x_1, respectively. C2 is not present in architecture 5, resulting in an inactive x_1 ($\delta_1 = 0$). This is also a good example of *declared* and *valid* design space size discrepancy [48]: the declared design space is given by the Cartesian product, in this example $n_{\text{declared}} = 4 \cdot 2 = 8$, whereas the valid design space is given by the number of feasible architectures, in this example $n_{\text{valid}} = 6$.

Table 4 For each architecture in Table 1, the associated design vector x, activeness information δ, and node existence. Selection choices C1 and C2 are mapped to x_0 and x_1, respectively.

	Design vector		Activeness		Node existence													
Arch.	x_0	x_1	δ_0	δ_1	N0	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12	N13
1	0	0	1	1		✓	✓	✓	✓			✓	✓	✓	✓			
2	0	1	1	1		✓	✓	✓	✓			✓				✓		
3	1	0	1	1		✓	✓	✓		✓	✓	✓	✓	✓	✓			
4	1	1	1	1		✓	✓	✓		✓	✓	✓	✓	✓	✓	✓		
5	2	0	1	0		✓	✓	✓			✓		✓	✓	✓			
6	3	1	1	1		✓	✓	✓				✓				✓		✓

Encoding and Decoding Connection Choices A Connection Choice Formulation (CCF) consists of connection (grouping) nodes, connection edges, exclusion edges, and node existence scenarios for a given connection choice node. Different CCF patterns are best encoded as design variables using dedicated encoding grammars [39, 40, 45]. This is due to two effects discussed by Selva [40]: bijectivity and non-degradedness.

Bijectivity relates to the difference in declared and valid design spaces: if this difference is large, it is more difficult to explore the design space for an optimization algorithm, because there is a low chance of generating a valid design vector when (randomly) searching the design space. We quantify this property using the discrete *imputation ratio*:

$$\mathrm{IR}_d = \frac{\prod_{j=1}^{n_{x_d}} N_j}{n_{\text{valid,discr}}} \tag{1}$$

where $n_{\text{valid,discr}}$ is the number of valid discrete design vectors, n_{x_d} is the number of discrete design variables, and N_j is the number of options for discrete variable j. An imputation ratio of 1 indicates a one-to-one mapping between design vectors and architectures (i.e. bijectivity), whereas values higher than 1 indicate this is not the case. The higher the value, the larger the discrepancy.

Non-degradedness can be quantified by looking at whether a small change in a design vector leads to a small change in what is represented by that design vector. Having this property improves optimizer performance [65]: optimization algorithms depend on it for local search and model building. In the case of connection choices, a design vector $\mathbf{x}$ represents a connection set, which we can represent as an n_{src} x n_{tgt} connection matrix M, where n_{src} and n_{tgt} represent the number of source and target nodes involved in a given connection choice, respectively. We define the distance correlation Dcorr, which correlates design vector distance to connection matrix distance:

$$\text{Dcorr} = \text{pearsonr}\left(\{d(\mathbf{x}, \mathbf{x}'), \dots\}, \{d(M, M'), \dots\}\right) \tag{2}$$

where pearsonr is the Pearson correlation coefficient, d is the Manhattan distance, and $\mathbf{x}$ and $\mathbf{x}'$ are two randomly sampled valid design vectors with M and M' their corresponding connection matrices. Dcorr = 1 indicates perfect correlation, meaning that a small change in a design vector indeed leads to a similarly small change in the connection matrix. Lower values indicate less correlation.

The goal is therefore for a given CCF, to select an encoder that minimizes IR_d and maximizes Dcorr. One possibility is to define a dedicated encoder for each architecture decision pattern. However, a CCF is more flexible than that (as discussed in Section III.A.1) and therefore the pattern encoders cannot cover all possibilities. To support all CCFs, we therefore define generic encoders. One way is to first enumerate all valid connection matrices M, and then encode these matrices as unique design vectors (M to $\mathbf{x}$ mapping), from which then design variables are defined. Design vectors are corrected in a greedy manner (i.e. one-by-one, starting from the left), and decoded by reverse lookup using the previously defined mapping. We denote this class of generic encoders as *eager* connection encoders, because they depend on the full enumeration of valid connection matrices in advance. If this is not possible due to time or memory limits, *lazy* encoders can be used instead: encoders that directly define design variables from the CCF, without needing all valid M. Since not all M is then available, correction is performed in a trial-and-error manner, repeatedly modifying a requested $\mathbf{x}$ until it represents a valid M. Finally, we also define *ordinal* encoders that simply map all valid M to an index, encoded as integers or a numeral system with some base (e.g. binary variables for base 2). Table 5 compares the different classes of connection choice encoders and their properties, in order of preference: pattern-specific encoders (fast correction/decoding, low memory usage), eager encoders (fast correction/decoding), lazy encoders (low memory usage), and ordinal encoders (fallback solution).

Table 5 Classes of connection choice encoders and their properties, in order of decreasing preference from left to right. Abbreviation: CCF = Connection Choice Formulation.

Encoder class	Pattern-specific	Eager	Lazy	Ordinal
Applies to	Patterns	All	All	All
Needs all $\boldsymbol{M}$	No	Yes	No	Yes
Encoding	Pattern-specific	Map $\boldsymbol{M}$ to $\mathbf{x}$	Based on CCF	Ordinal enumeration
Decoding	Pattern-specific	Reverse $\boldsymbol{M}$ lookup	Encoder-specific	$\boldsymbol{M}$ indexing
Correction	Pattern-specific	Greedy	Trial-and-error	Clipping
Encoding time	Very fast	Slow	Fast	Medium
Decoding time	Very fast	Fast	Slow	Very fast
Correction time	Very fast	Fast	Slow	Very fast
Memory usage	Low	High	Low	Low

Design Space Insights and Generating Architecture Instances Encoding a DSG into a set of design variables is done by first encoding selection choices, then encoding connection choices, and finally adding design variables defined by design variable nodes. Decoding is done in the same order: decode and resolve selection choices, decode and resolve connection choices, set active design variable values, and impute inactive design variables (set inactive discrete variables to 0, continuous variables to mid-bounds). If the complete selection choice encoder is used, it is possible to generate $x_{\text{valid,discr}}$. This enables usage of the hierarchical sampling and correction algorithms developed in [48]. Additionally, it allows the problem-level calculation of the number of valid architectures, and thereby the problem-level IR.

B. The ADSG: Applying the DSG for System Architecting

Generic nodes in the DSG represent elements that can be included in architecture instances, however by themselves they have no semantic meaning. We now present the Architecture Design Space Graph (ADSG): a database of node types and allowable connections that assigns meaning to the generic DSG nodes for use in a system architecting context. This enables function-based architecture definition, as originally presented in [9]. Nodes are defined based on the subdivision of architecting tasks defined by Crawley et al. [1]: function decomposition and function-to-component (form) allocation, component characterization, and component connection. Boundary functions, functions that act at the system boundary and deliver the main value to the system stakeholders [29], are used as start nodes. Function-to-component allocation decomposes functions, assigns functions to components (function fulfillment), and derives new functions from component selection (function induction). Component characterization represents component-level choices, including the number of component instances, design variable value allocation, and attribute selection. Component connection involves the definition of port connectors for each associated component instance, and the connection of output ports to input ports.

Table 10 lists all ADSG node types and possible derivation edge connections. Functions specify what the system should do; components represent form and define how the functions are fulfilled. A function-to-component derivation edge represents function fulfillment; the opposite represents function induction: the addition of a function because that component has been selected [29]. Several complexity management nodes are provided: concept nodes for mapping a solution-neutral to a solution-specific function, decomposition nodes for 1-to-n mapping of function nodes, and the newly added non-fulfillment and multi-fulfillment nodes. Selection and connection choices are automatically added based on graph patterns presented in Table 11. A selection choice is for example added when a function connects to multiple components, representing a function fulfillment choice. For more detailed discussion of the semantic meaning of remaining nodes in a system architecting context the reader is referred to [9]. Figure 12 shows an example ADSG.

C. ADORE: Editing and Exploring Architecture Design Spaces

The ADSG is implemented in ADORE (Architecture Design and Optimization Reasoning Environment) [56], an in-house Python tool developed by the DLR Institute of System Architectures in Aeronautics. ADORE is built on three pillars that enable the definition of architecture design spaces and solving SAO problems:

- A web-based graphical user interface (GUI) for editing and inspecting the ADSG.
- Python and file-based interfaces for connecting to evaluation code.
- Interfaces for connecting to optimization algorithms.

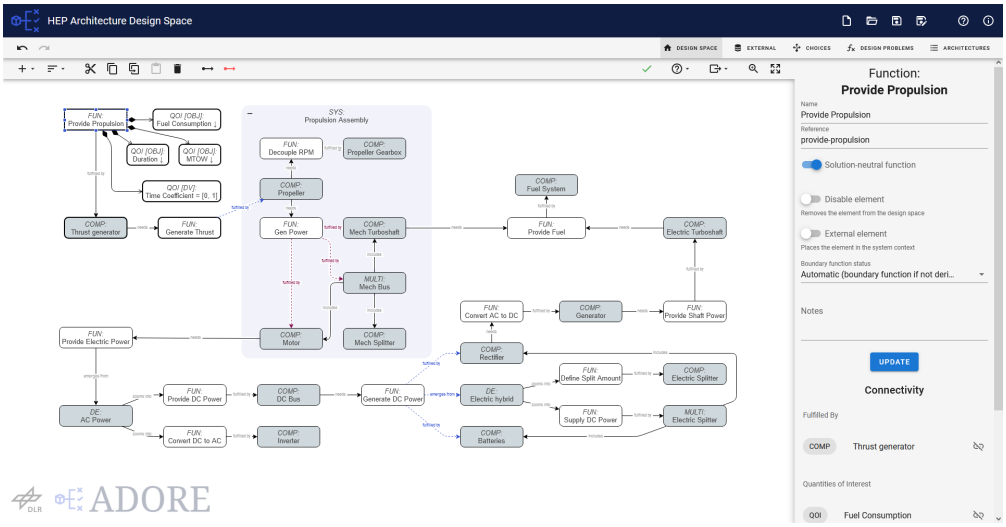


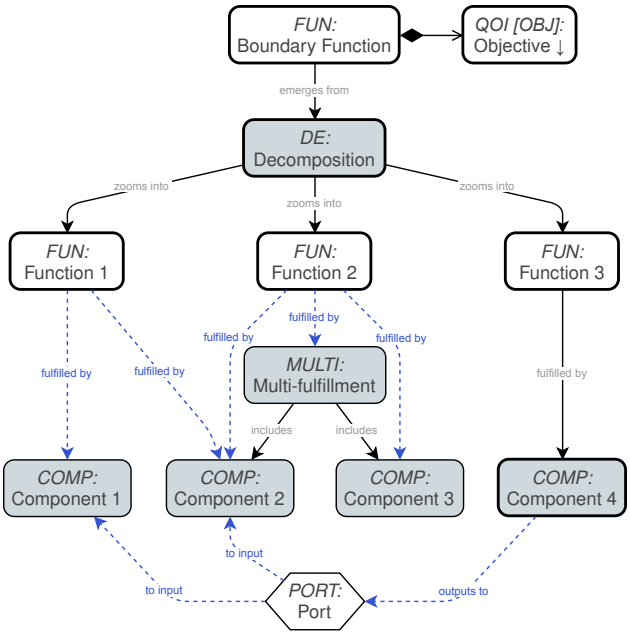
Fig. 3 ADORE web-based graphical user interface, showing the design space canvas in system view.

The architecture design space is defined in an ADORE model, from which the ADSG is constructed. An ADORE model represents the same concepts, however with several differences to improve modeling experience. The *web-based GUI* (see Figure 3) contains a canvas for editing the ADORE model, a list showing all defined architectural choices and design space statistics, a page for defining optimization problems, and a page for viewing and manually creating architecture instances. The ADORE model shows architectural choices by blue dashed arrows (instead of choice nodes in the ADSG) and defines several views: a system view showing functions, function derivation elements and ports; a component view showing component-level elements; and a port view showing a port and its port connectors. Metrics, design variables, and static inputs are defined using Quantity of Interests (QOIs), enabling flexible switching between input or output roles. This is useful, as during design space definition it may not be known whether some value is an input to or an output from the architecture evaluation process. The ADORE model also adds the capability for modeling subsystems: recursive groups of elements, where the number of subsystem instances may also be an architectural choice. Each subsystem instance contains copies of the original elements, including choices, and copied choices are independent of each other. Figure 4 shows the ADORE model equivalent to the ADSG shown in Figure 12.

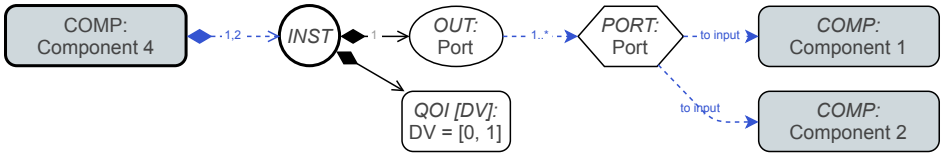
Connection to *evaluation* code can be established in several ways, depending on what is more appropriate considering available analysis tools, implementation environment, and/or programming skills. The most flexible way to implement evaluation is by implementing an evaluation function, a function returning numerical values for all included output QOIs for a given architecture instance (provided as an ARCHITECTURE class), directly in Python. Python-based evaluation can be supported by the *Class Factory Evaluator (CFE)* [62]. With the CFE, the user can define rules for instantiating objects based on selected architecture elements, thereby making it easy to use object-oriented input definitions for evaluation functions (see Section IV.C for an application). Another aid for Python-based evaluation is the *Supplementary Design Space Graph (SupDSG)*: a DSG with choices mapped to choices in the ADORE model. The SupDSG enables modeling variability in graphs where edges and nodes do not necessarily have system architecture semantics. For an example application of the CFE and SupDSG, see Section IV.C. For integration with external platforms, it is also possible to use a *file-based* evaluation interface. Here, generated architectures are serialized as XML, JSON, or RDF, which are then used as input to some externally-integrated evaluation function.

After the design space has been modeled and the evaluation function has been connected, the optimization problem can be formulated and executed. Running the optimization problem is done by connecting to SBArchOpt[†] (Surrogate-Based Architecture Optimization), an open-source Python library that provides an API for defining SAO problems and connects to various optimization algorithms and frameworks [66]. SBArchOpt adds SAO capabilities on top of pymoo[‡], such as hierarchical sampling and correction, and an implementation of an SBO algorithm for architecture optimization: ArchSBO [48, 49].

[†]https://sbarchopt.readthedocs.io/
[‡]https://pymoo.org/



(a) System view



(b) Component view (Component 4)

Fig. 4 ADORE model showing a design space with a boundary function (which is also the start function) with an objective, a decomposition into 3 lower-level functions, selection choices (2x function fulfillment, 1x component instantiation), instance-level design variables, and a port connection choice. Figure 12 shows the equivalent ADSG. Blue-dashed arrows indicate architectural choices.

D. Section Conclusions

The preceding sections have introduced the DSG for modeling architectural choices using a directed graph and encoding these as an SAO problem, the ADSG for adding meaning to nodes in a system architecting context, and ADORE for modeling an ADSG in a GUI and connecting to evaluation code and optimization algorithms. Figure 13 shows the interaction between the three layers, and how the layers are involved in the SAO loop.

IV. Demonstration: Architecture Optimization with ADORE

In this section, we demonstrate the ADSG and ADORE by three test problems: the design of a multi-stage launch vehicle demonstrating selection choices and evaluation using file-based dynamic MDAO (Section IV.A), the design of a guidance, navigation and control system demonstrating connection choices (Section IV.B), and the design of a jet engine demonstrating the Class Factory Evaluator (CFE), Supplementary Design Space Graph (SupDSG), and MDAO for evaluation (Section IV.C). Table 6 presents problem statistics of the test problems, along with application cases published since the original ADSG publication [9].

A. Multi-Stage Launch Vehicle Architecture

The multi-stage launch vehicle SAO problem was originally developed to test dynamic MDAO formulation in [63, 72], however also from an SAO point-of-view it is an interesting problem. It involves the choice of number of stages, several stage-level choices (number of engines, engine types, and stage length) and rocket geometry choices (head shape and

Table 6 Overview of ADSG/ADORE application cases. Abbreviations: n_{x_d} = discrete variables, n_{x_c} = continuous variables, n_f = objectives, n_g = constraints, IR = imputation ratio, GNC = guidance, navigation & control, CFE = class factory evaluator, DoE = Design of Experiments, SupDSG = supplementary design space graph, FCS = Flight Control System.

Problem	n_{x_d}	n_{x_c}	n_f	n_g	$n_{\text{valid,discr}}$	IR	Optimizer	Translation	Evaluation
Apollo mission [9]	8		2		108	5.3	Enumeration		Python
Supersonic business jet [56]		9	2	11	1	1	ArchSBO [48]	MultiLinQ [56]	MDAO
Business jet family [67]	10	9	2	6	1024	1.5	SEGOMOE [68]	MultiLinQ	MDAO
Hybrid-electric propulsion [62]	8	14	3		310	79.4	ArchSBO	CFE [62]	MDAO [61]
Landing gear braking [64]	13				100 352	1.3	DoE	CFE	ASSESS
Space mission [69]	2	10	2	2	25	1.7	NSGA-II	CFE	Python
FCS (spoilers) [70]	22		1	3	139 260	11 700	NSGA-II	CFE	ACOBS
FCS (spoilers & ailerons) [71]	32		3	3	< 6.3e12	—	ArchSBO	CFE	MDAO
Launch vehicle (Section IV.A)	8	6	2	2	18 522	2.5	ArchSBO, NSGA-II	XML	Dynamic MDAO
GNC (Section IV.B)	33		2		79 091 323	367	ArchSBO, NSGA-II		Python
Jet engine (Section IV.C)	15	9	1	5	70	9.1	ArchSBO	CFE, SupDSG	MDAO [60]

length-to-diameter ratio). It is a multi-objective problem, with the goal to maximize payload mass and minimize cost for a given target orbit altitude, subject to structural (max-Q) and payload volume constraints.

Figure 5 presents the ADORE model, showing the decomposition of the main function "Carry Payload to Space", the associated objectives, and the rocket stage subsystem. The Stage subsystem contains the rocket body component with the stage length design variable, and the engine assembly with the choice of engine type. Both the Stage and Engine Assembly subsystems can be instantiated 1, 2 or 3 times. Engine selection choices are linked, shown by purple dashed edges [62], because in the original problem formulation it is not possible to select different engine types for each stage. In [63, 72], architecture instances are evaluated with a file-based dynamic MDAO workflow. The purpose of that work was to demonstrate the formulation of such a dynamic workflow that for example switches between liquid and solid propulsion calculations based on selected engine types, repeats stage-level disciplines based on the selected number of stages, and dynamically (re)connects inputs and outputs based on available state variables. In this work, we evaluate architectures using the Python-based evaluation code available in SBArchOpt, which functionally represents the same dynamic MDAO workflow of the original implementation.

We compare performance of the ADORE formulation both for the fast and the complete selection choice encoders (denoted "ADORE Fast" and "ADORE Complete", respectively). The ADORE formulations are compared against a manual formulation available as LCRocketArch in SBArchOpt, denoted as "Manual". Table 7 lists problem statistics. All formulations result in the same design variables and IR, CR, and CRF because of the low degree of hierarchical coupling: activeness is only determined by the number of stages and head shape design variables. We also compare the Correction Time (CT), the average time needed to correct and impute one design vector, needed for the different formulations; this is relevant because this operation may be called orders of magnitude more often than function evaluation, for example when generating a DoE or when searching for infill points in SBO. CT for the Manual and ADORE Complete formulations is significantly lower than ADORE Fast, because for the ADORE Fast formulation $x_{\text{valid,discr}}$ is not available and therefore a trial-and-error approach has to be used. ADORE Complete, however, still requires some model-parsing overhead, so Manual has the lowest CT.

Table 7 Multi-stage launch vehicle problem formulations. The problem contains 18522 valid discrete design vectors. Abbreviations: n_{x_d} = discrete variables, n_{x_c} = continuous variables, IR = imputation ratio, CR = correction ratio, CRF = correction fraction, CT = correction time.

Formulation	n_{x_d}	n_{x_c}	IR	CR	CRF	CT [ms]
Manual	8	6	3.7	1.6	38%	0.21
ADORE Fast	8	6	3.7	1.6	38%	17
ADORE Complete	8	6	3.7	1.6	38%	7.0

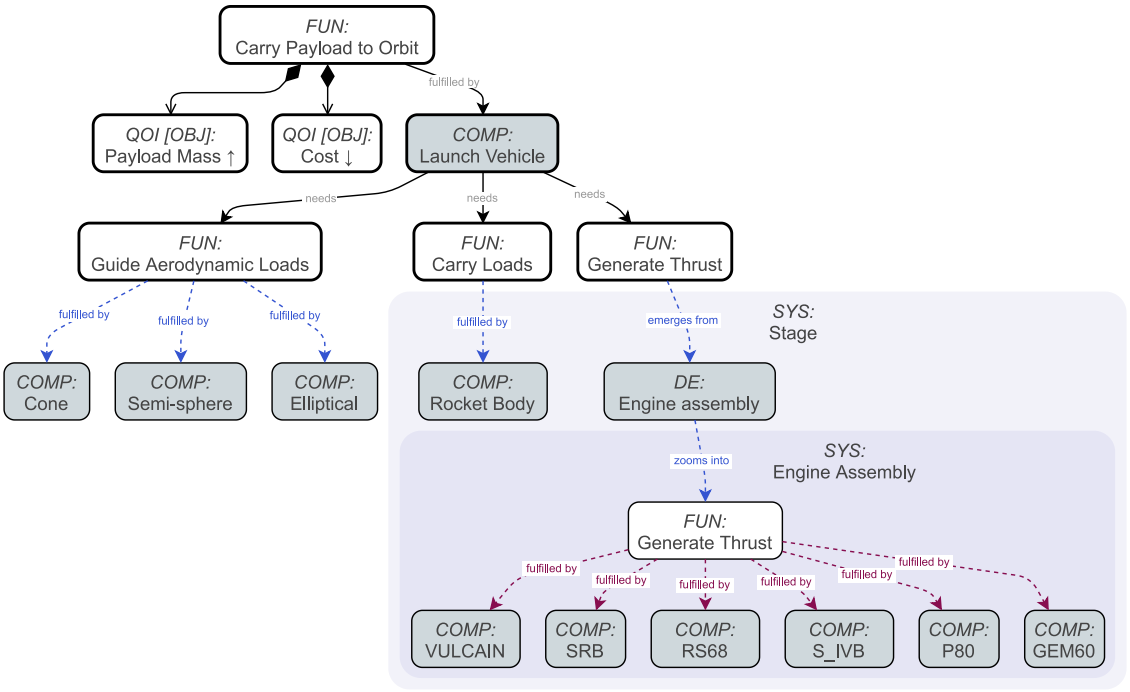
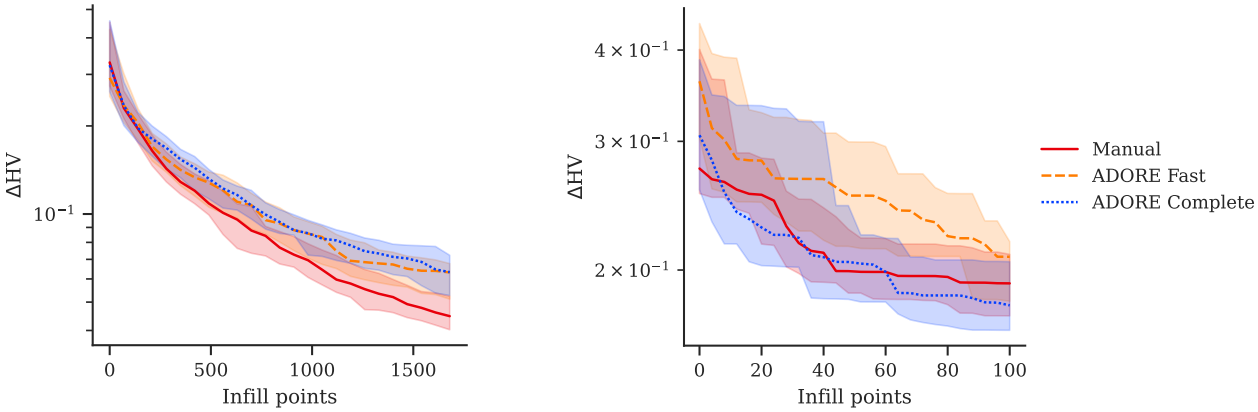


Fig. 5 ADORE model showing the design space of the launch vehicle problem in system view.

The different formulations are solved using NSGA-II, a multi-objective evolutionary algorithm (EA), and ArchSBO [48], a Bayesian Optimization (BO) algorithm tailored for SAO. Both algorithms start from an initial DoE of 70 points; NSGA-II is executed for 25 generations (population size 140) and 40 repetitions, ArchSBO is executed for 100 infill points with a batch infill size of 4, and 12 repetitions. Performance is compared using ΔHV (lower is better), representing the difference to the known Pareto front hypervolume, as a function of number of infill points (due to the assumption of expensive evaluation). Figure 6 presents optimization results. For NSGA-II, the Manual formulation performs best and the two ADORE formulations perform similarly. The difference in performance is due to a different arrangement of design variables, leading to slight differences in the initial population. For ArchSBO, the ADORE Fast formulation is performing worse than the Manual and ADORE Complete formulations.



(a) NSGA-II **(b) ArchSBO**

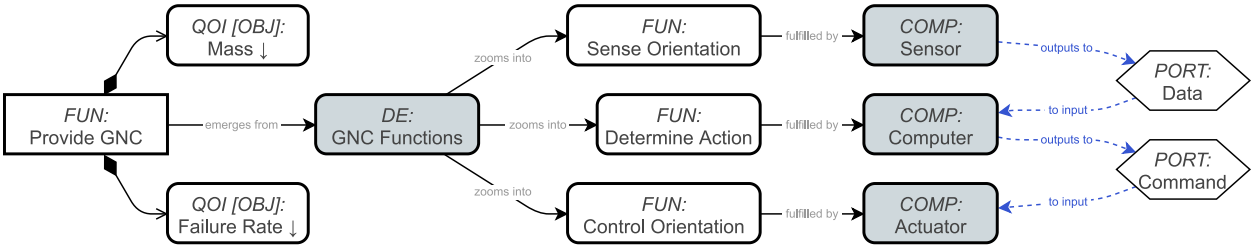
Fig. 6 Multi-stage launch vehicle problem solved using two optimization algorithms.

B. Guidance, Navigation and Control Architecture

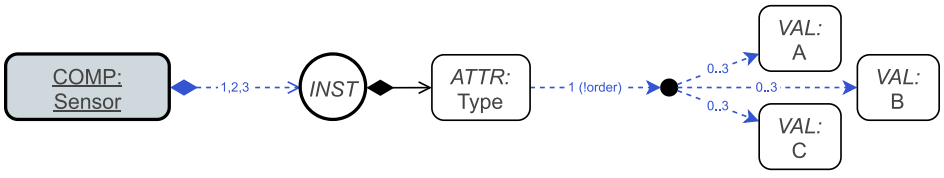
The GNC (Guidance, Navigation & Control) problem [1, 39] features the definition of an architecture connecting sensors (S) to flight computers (C), and flight computers to actuators (A). Each object (S, C, or A) can be instantiated 1, 2 or 3 times, and for each instance there are three types available (A, B or C) with a different masses and reliabilities associated to each (in general, with increasing reliability as mass increases). The connections from S to C and C to A are architectural decisions, with each connection increasing reliability, and a constraint that no object is left unconnected. For each object, only unordered combinations of types can be assigned, as permutations of types lead to the same architecture if also the associated connections are permuted (for example, AB sensors connected to BC computers represent the same architecture as BA sensors connected to CB computers if the connections are reversed as well). The problem objectives are mass, calculated from the sum of selected object masses, and system-level reliability, calculated from a failure-tree approach assuming that the system does not fail as long as at least one S-C-A path is still operational (i.e. the objects and connections have not failed). We note that our interpretation of the problem is slightly different from [1, 39], so results cannot be compared directly.

In ADORE, we model the architecture design space by decomposing the boundary function "Provide GNC" into sensing (fulfilled by sensors), determining action (fulfilled by computers), and controlling (fulfilled by actuators). Ports are used to model component connections: "Data" represents the sensor to computer connection, "Command" the computer to actuator connection. Figure 7a show the system view, including the two system-level objective mass and failure rate, both to be minimized. Each component has 1, 2 or 3 instances, and an attribute specifying the type, see Figure 7b. Attributes are modeled as connection choices, "connecting" from attribute to value: in this case each component instance has an attribute needing exactly 1 connection, and each value can be connected to between 0 and 3 times. On the attribute side, order is set to irrelevant (shown by "!order"), effectively grouping outgoing connections and ensure only unordered combinations can be selected. Connections are modeled using ports, with the only constraint being that each connector has at least one connection (shown by "1..*"), see Figure 7c.

For selection choice encoding (use for component instantiation) we compare the fast and complete selection choice encoders (denoted "ADORE Fast" and "ADORE Complete", respectively). In addition to these two ADORE formulations, the GNC problem is also implemented with a manual encoding (called "Manual") and only using connection choice encoders (called "Encoded"). The Manual and Encoded formulations are available in SBArchOpt as GNC and AssignmentGNC, respectively. Table 8 presents statistics of the GNC problem, including and excluding



(a) System view



(b) Component view (Sensor)

(c) Port view (Data)

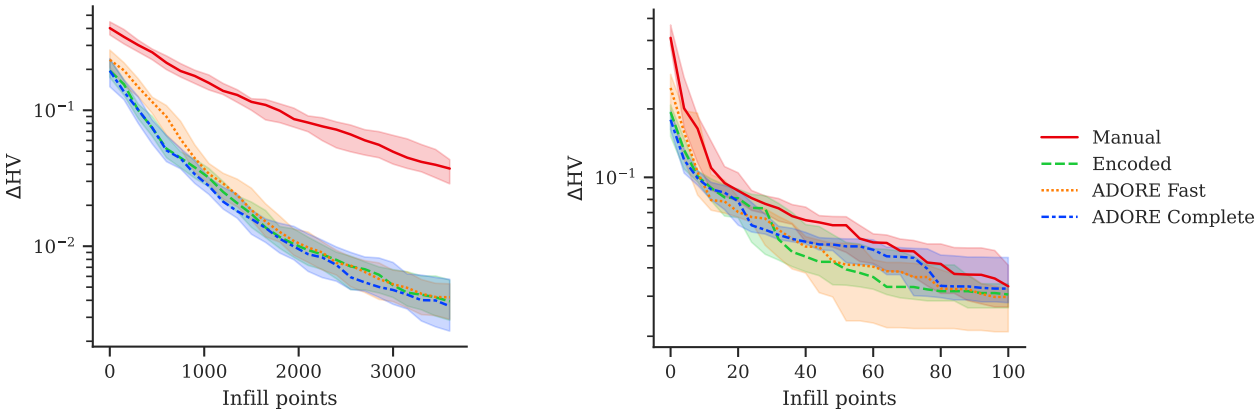
Fig. 7 ADORE model showing the design space of the GNC problem in system view, component view and port view. All components and ports have a similar definition.

Table 8 GNC problem formulations. Abbreviations: n_{x_d} = discrete variables, IR = imputation ratio, CR = correction ratio, CRF = correction fraction, CT = correction time.

	Including actuators			Excluding actuators				
Formulation	n_{x_d}	IR	CT [ms]	n_{x_d}	IR	CR	CRF	CT [ms]
Manual	30	1761	4.7	17	113	17.2	60%	1.4
Encoded	33	367	38	19	39.5	6.0	49%	12
ADORE Fast	30	23460	650	17	632	—	—	102
ADORE Complete	33	367	62	19	39.5	6.0	49%	17
$n_{\text{valid,discr}}$		79 091 323			29 857			

actuators. It shows that the ADORE Complete formulation obtains the same problem definition as the Encoded formulation. The Manual formulation results in a higher IR and CR compared to the Encoded and ADORE Complete formulations, showing that automatically choosing the connection choice encoders improves problem formulation. ADORE Fast formulation results in a very high IR, because the fast selection choice encoder is not able to correctly determine all connector node existence scenarios, which results in less efficient connection choice encoding. Manual formulation has the fastest CT as it features problem-specific code tailored to the formulation. Encoded and ADORE Complete formulations are slightly slower, as they depend on greedy correction that uses design vector lookup. ADORE Fast correction is slowest as it depends on trial-and-error correction. Except ADORE Fast, the problem formulations excluding actuators do provide $x_{\text{valid,discr}}$, which enables the use of the hierarchical sampling algorithm presented in [48].

The different formulations (including actuators) are solved using NSGA-II and ArchSBO [48]. Both algorithms start from an initial DoE of 150 points; NSGA-II is executed for 25 generations (population size 150) and 40 repetitions, ArchSBO is executed for 100 infill points with a batch infill size of 4, and 12 repetitions. Figure 8 presents optimization results. It shows that for NSGA-II, the Manual formulation performs significantly worse than the other formulations, which all perform similarly. For ArchSBO, all formulations perform similarly, with the Manual formulation performing slightly worse than the others. It can be concluded that the ADORE formulations all result in problem formulations that can be solved by EA and BO algorithms, and that they perform as well as the Encoded formulation and better than the Manual formulation. ADORE Complete outperforms ADORE Fast in terms of IR, correction time, and is similar in optimizer performance.



(a) NSGA-II **(b) ArchSBO**

Fig. 8 GNC problem (including actuators) solved using two optimization algorithms.

C. Jet Engine Architecture

The jet engine architecture problem [60] features the selection and sizing of jet engine components. Here, we use the simple problem definition presented in [60], which is a single-objective problem minimizing Thrust-Specific Fuel Consumption (TSFC) subject to several feasibility constraints. Architectural choices include adding a fan (turbofan architecture) or not (turbojet architecture), the number of compressor/turbine stages, whether bypass and core flows are mixed before flowing out, whether a gearbox is added between the fan and low-pressure shaft, and the locations of bleed air and power offtakes. Continuous sizing variables include bypass ratio, fan pressure ratio, compressor pressure ratios, gearbox ratio and shaft rpm's.

Figure 9 shows the ADORE model with three boundary functions: Generate Thrust, Provide Bleed Air, and Provide Power. The Generate Thrust function has the TSFC objective and weight metrics associated to it, and its fulfillment represents the choice whether to include a fan or not. The nozzle mixing choice is represented by the fulfillment of the Exit Core/Bypass Flow functions. Incompatibility constraints are used to model the constraint that either a mixed nozzle is selected, or both the (core) nozzle and bypass nozzle. The (core) nozzle and mixed nozzle both need the Energize Air function, which derives the remaining components in the engine core. The Compressor, Shaft and Turbine components include instantiation choices (1, 2 or 3), which are linked by a choice constraint ensuring that the number of instances match. The Provide Bleed Air and Power functions are fulfilled by connecting a Bleed Air Duct and Generator to one of the Compressors and Shafts, respectively. The gearbox choice is an example of non-fulfillment: the function Uncouple Fan RPM represents an "improvement" function which can either be fulfilled by the gearbox, or it can be left unfulfilled. Sizing variables (e.g. bypass ratio, compressor ratios) are modeled as design variable QOIs of the respective components.

Jet engine architecture instances are sized and evaluated using the framework presented in [60], which constructs an MDAO problem using pyCycle [73] and OpenMDAO [74] based on a set of ArchElement classes that represent engine elements. Each class contains several properties that provide input to the analysis, some of which are also architecture-level sizing variables, such as pressure ratios and rpm's. The thermodynamic cycle analysis is based on coupling a series of airflow elements, with each element having airflow properties (e.g. pressure, temperature) both

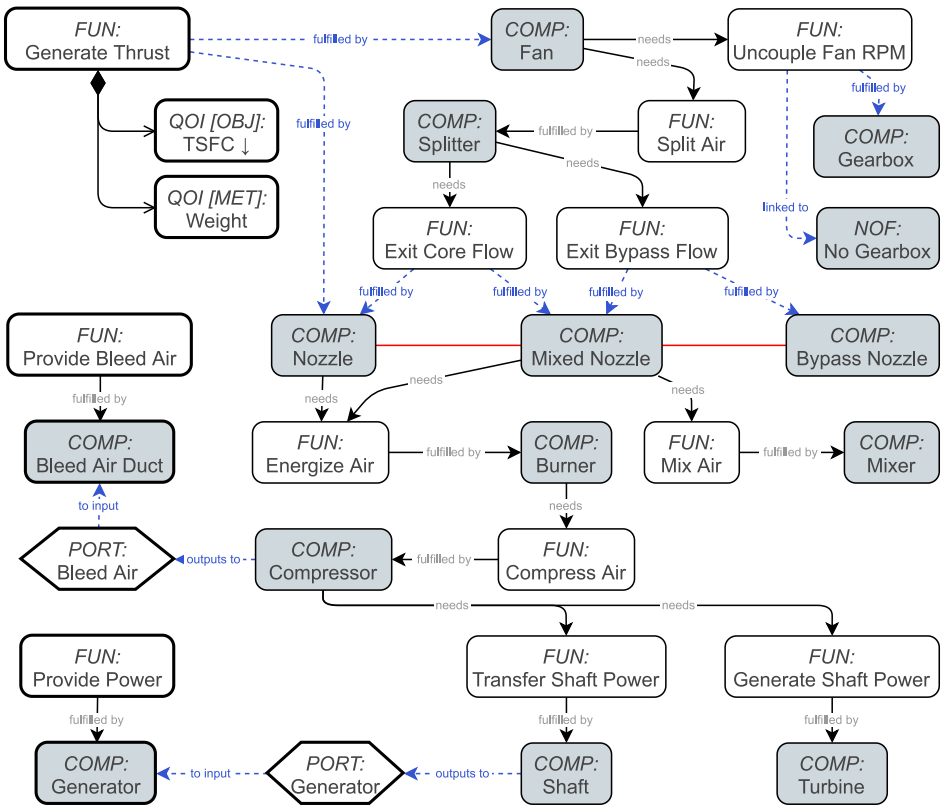


Fig. 9 ADORE model showing the design space of the jet engine problem in system view.

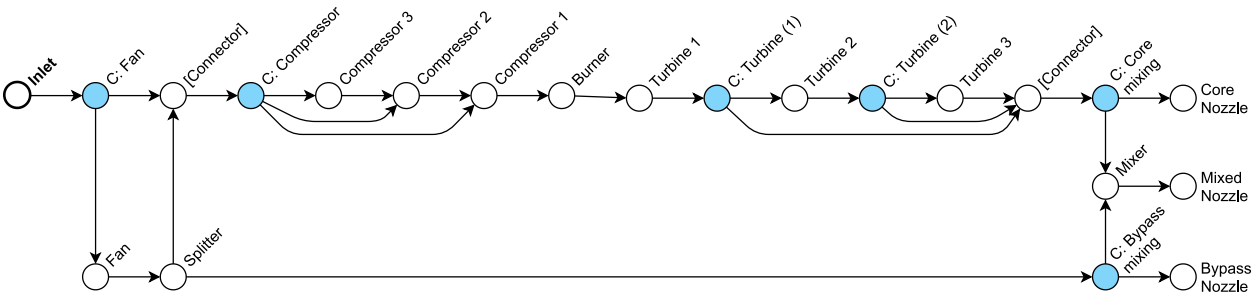


Fig. 10 Supplementary Design Space Graph (SupDSG) modeling airflow variability for the jet engine problem. Choice nodes are shown in blue, edges represent derivation edges. "Inlet" is the start node.

as analysis input and output. To automatically construct the MDAO problem, the connection sequence of elements should therefore be defined. In [60], this is defined implicitly as part of the ArchElement instantiation code, however this makes it difficult to trace exactly which elements are connect to each other for different architecture instances. To mitigate this, we use a Supplementary Design Space Graph (SupDSG). A SupDSG is an extension of the ADSG that allows modeling graph variability in contexts other than system architecting: for this problem, we use it to model variability in the airflow as shown in Figure 10. Choice nodes are mapped to architecture choices in the ADORE model: Fan, Compressor and Core/Bypass mixing choices are mapped 1-to-1 to their ADORE equivalents; the turbine choice is divided in two, because otherwise it is not possible to maintain the same direction for derivation edges and airflow. All nodes except choice and "[connector]" nodes are mapped to ArchElements, thereby specifying the airflow path for each architecture instance.

We compare the ADORE model formulated with the fast and complete selection choice encoders (denoted "ADORE Fast" and "ADORE Complete") with the manual formulation from [60] (denoted "Manual"). The Manual formulation is available in SBArchOpt as SimpleTurbofanArch. Table 9 lists statistics, showing that the Manual and ADORE Complete formulations are similar, and the ADORE Fast formulation leads to a higher IR. Manual and ADORE Complete CT values are significantly lower than ADORE Fast, for the same reasons as for the rocket problem ($x_{\text{valid,discr}}$ availability). Optimizer performance is compared for ArchSBO, the hierarchical BO algorithm presented in [48]. The algorithm is run with an initial DoE of 75 points, 200 infill points, 4 points evaluated in parallel, and 12 repetitions. Figure 11 presents optimization results, showing that the Manual and ADORE Complete formulations perform similarly, and ADORE Fast performs slightly worse.

Table 9 Jet engine problem formulations. The problem contains 70 valid discrete design vectors. Abbreviations: n_{x_d} = discrete variables, n_{x_c} = continuous variables, IR = imputation ratio, CR = correction ratio, CRF = correction fraction, CT = correction time.

Formulation	n_{x_d}	n_{x_c}	IR	CR	CRF	CT [ms]
Manual	6	9	3.9	2.1	55%	9.0
ADORE Fast	7	9	9.3	2.1	33%	87
ADORE Complete	6	9	4.6	2.1	55%	17

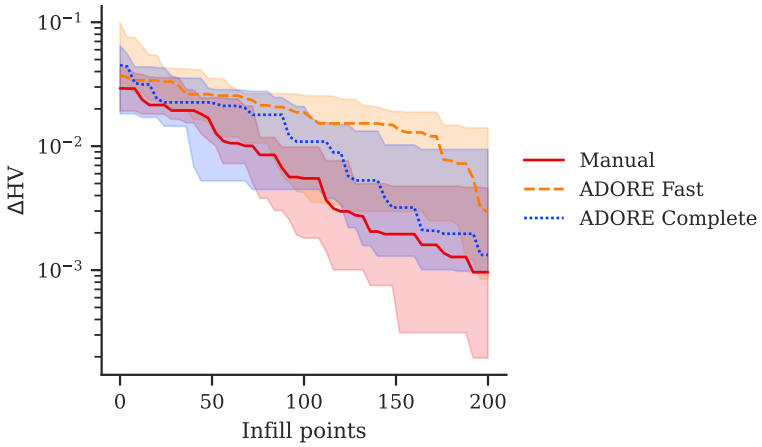


Fig. 11 Jet engine problem solved using the ArchSBO algorithm.

V. Conclusions and Outlook

The Design Space Graph (DSG) is a directed graph that models node and choice hierarchy using selection choices, and connection patterns using connection choices. Selection choices are encoded using a complete encoder, enabling full enumeration of all valid design vectors, or a fast encoder that needs less time and memory to encode. Connection choices are encoded by pattern-specific, eager, lazy or ordinal encoders. For each Connection Choice Formulation (CCF), the encoder is automatically selected based on two metrics (IR_d and Dcorr). Incompatibility constraints and choice constraints can be used to further constrain node selection and architectural choices. The DSG can then formulate a design problem by encoding architectural choices as design variables and specifying objectives and constraints using metric nodes. The Python implementation of the DSG is available open-source as ADSG Core.

To apply the DSG in a system architecting context, the Architecture Design Space Graph (ADSG) defines node types with semantic meaning. For example, function, component, decomposition, multi-fulfillment, component instance, and port nodes. A function connected to a component represents function fulfillment; the opposite represents function induction. To create an ADSG, the ADORE modeling environment is used. ADORE provides a web-based GUI for creating ADORE models (from which the ADSG is constructed), interfaces for implementing performance evaluation functions, and an interface for connecting to optimization algorithms provided by SBArchOpt.

The ADSG is demonstrated by three demonstration cases: the design of a multi-stage launch vehicle (18.5k architectures in the design space, demonstrating evaluation using dynamic collaborative MDAO), a guidance, navigation and control system (79m architectures in the design space, demonstrating connection choice encoders), and a jet engine architecture (70 architectures in the design space, demonstrating the Class Factory Evaluator and Supplementary Design Space Graph). For each problem, formulations modeled in ADORE and encoded with the fast and complete encoders are compared with manual formulations of the same problem. For all problems it is demonstrated that the ADORE formulation results in similar optimizer performance as the best manual formulation if the complete selection choice encoder is used. If the fast encoder is used, performance is reduced slightly. It can therefore be concluded that the ADSG as implemented in ADORE can be used to define optimization problems using system architecting terminology (functions, components, etc.) that are as solvable as manually-formulated optimization problems (i.e. defined in terms of design variables, objectives, constraints, etc.).

Support for very large design space should be improved, so that hierarchical sampling and correction can also be used in cases where currently the complete selection choice encoder is not available. In future work, the method should be applied to more system architecture problems, such as on-board systems or System-of-Systems problems, with more design variables and objectives than currently tested. Especially high-dimensional optimization problems (e.g. with hundreds of design variables) are expected to pose problems for current SBO algorithms. ADORE should be integrated with existing MBSE methods, enabling seamless transition from requirements engineering to system architecting, to detailed design. A good candidate for this is the SysMLv2 language, which already supports variability modeling, however might benefit from full SAO capabilities. Finally, collaborative MDAO should be applied to architecture performance evaluation. The first steps towards this has been presented in [63], however integration should be made easier by providing a generic mapping interface from generated architecture instances to the data model used in the MDAO workflow.

Appendix

Table 10 ADSG node types and allowed incoming/outgoing derivation edge connections. Abbreviations: FDN = function derivation node (node in the function allocation task other than the function node), F/M = function or multi-fulfillment node, QOI = Quantity of Interest (MET, DV, or INP), CCN = component characterization node (QOI or ATTR), C/I = COMP or INST.

Task	Node	Symbol	Node type	Incoming	Outgoing
Function allocation	Function	FUN	Generic	0+ FDN	1+ FDN, 0+ QOI
	Component	COMP	Generic	1+ F/M	0+ FUN or CCN, 1+ INST
	Concept	CON	Generic	1 F/M	1 FUN
	Decomposition	DE	Generic	1 F/M	1+ FUN
	Non-fulfillment	NOF	Generic	1 F/M	
	Multi-fulfillment	MULTI	Generic	1+ FUN	1+ FDN
Component characterization	Component instance	INST	Generic	1 COMP	0+ CCN, 0+ IN/OUT
	Attribute	ATTR	Generic	1 C/I	1+ VAL
	Attribute value	VAL	Generic	1 ATTR	
	Metric	MET	Metric	1 C/I or FUNC	
	Design variable	DV	Design variable	1 C/I or FUNC	
	Static input	INP	Generic	1 C/I or FUNC	
Component connection	Output port	OUT	Connector	1 INST	Optional GRP
	Input port	IN	Connector	1 INST	Optional GRP
	Port grouping	GRP	Connector grouping	1 IN/OUT	

Table 11 ADSG architecture choices. Choices are inserted for each matching incoming-outgoing pair.

Choice	Type	Incoming	Outgoing
Function fulfillment	Selection	1 FUN	2+ FDN
Component instantiation	Selection	1 COMP	2+ INST
Attribute value selection	Connection	1 ATTR	1+ VAL
Port connector instantiation	Selection	1 INST	2+ IN/OUT
Port connection	Connection	1+ OUT	1+ IN

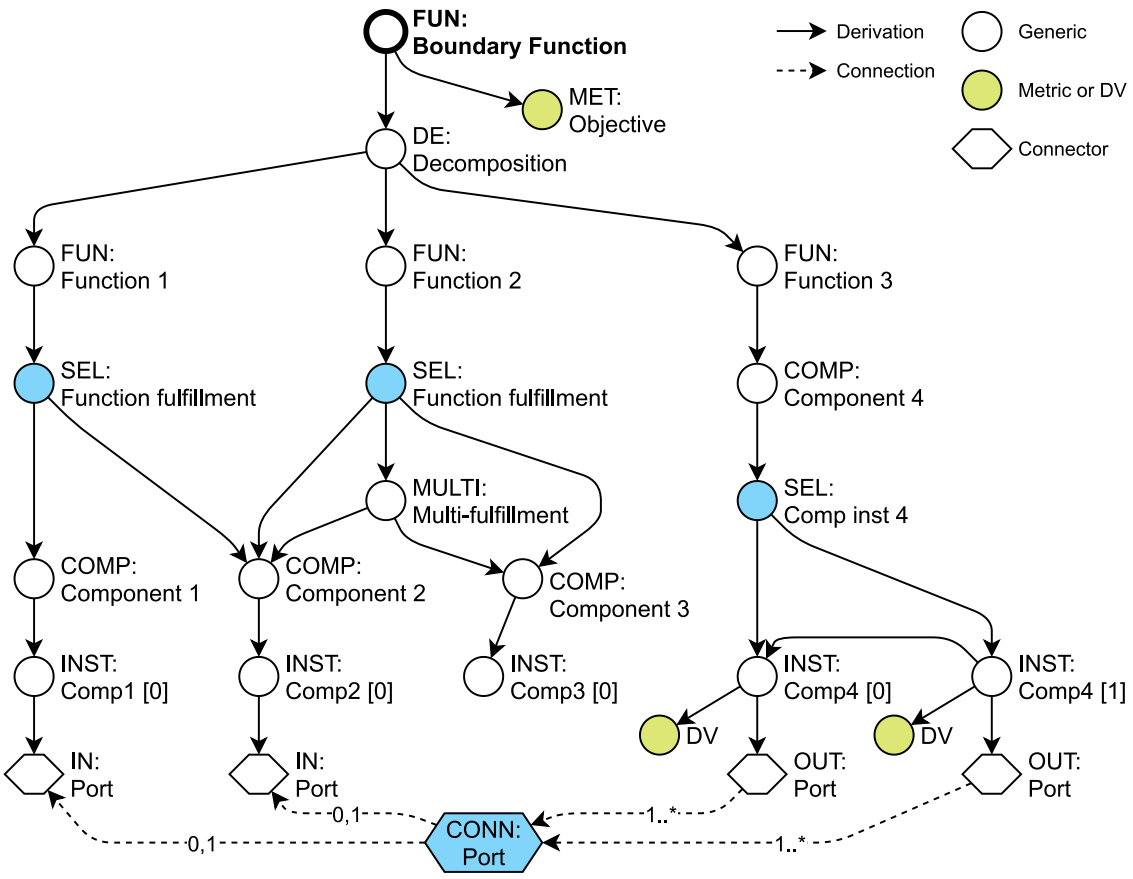


Fig. 12 ADSG example, showing a design space with a boundary function (which is also the start function) with an objective, a decomposition into 3 lower-level functions, selection choices (2x function fulfillment, 1x component instantiation), instance-level design variables, and a port connection choice. Figure 4 shows the equivalent ADORE model.

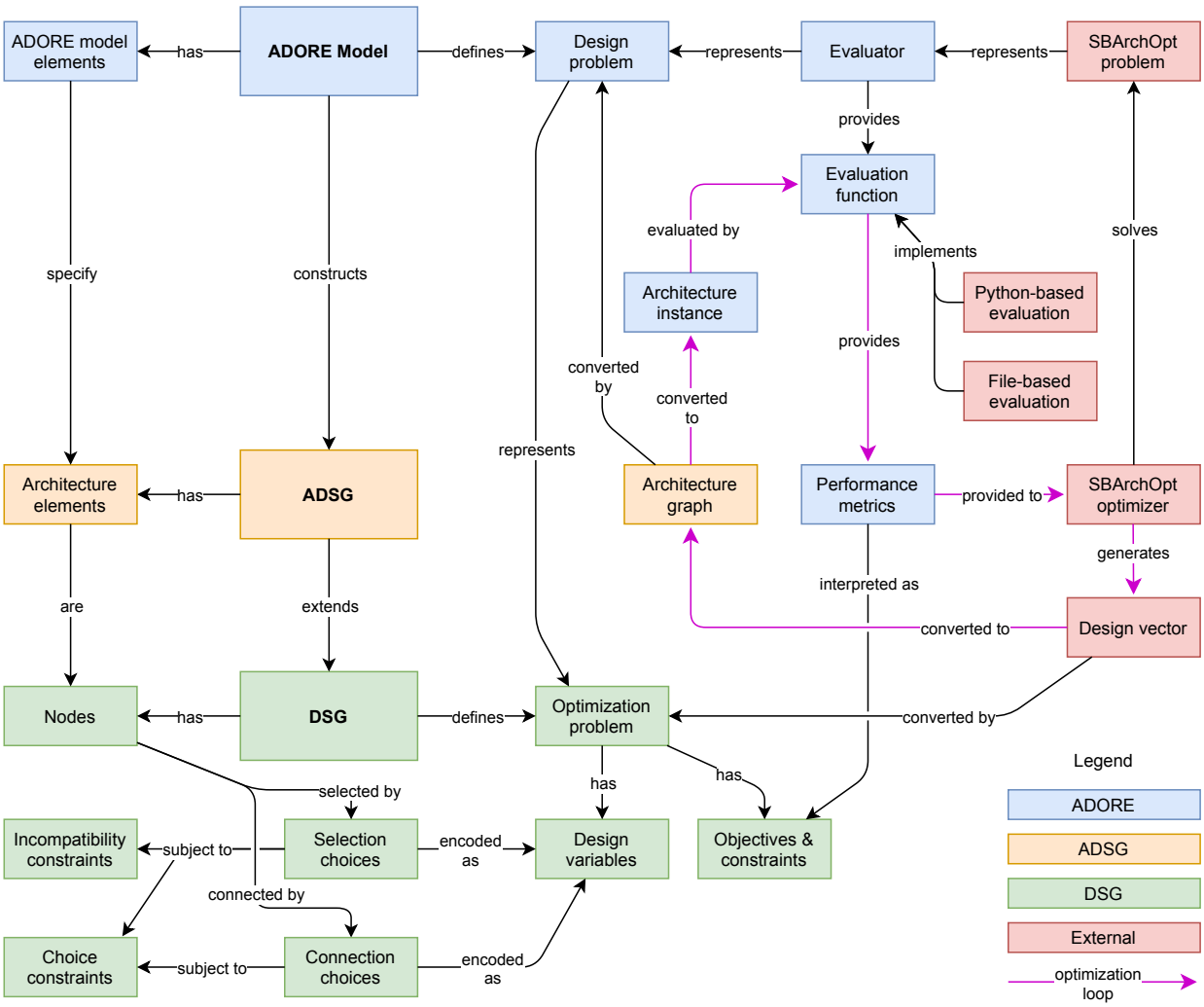


Fig. 13 Connection between DSG, ADSG and ADORE layers and how these are involved in the SAO loop.

Acknowledgments

The research presented in this paper has been performed in the framework of the COLOSSUS project (Collaborative System of Systems Exploration of Aviation Products, Services and Business Models) and has received funding from the European Union Horizon Europe Programme under grant agreement n° 101097120.

References

[1] Crawley, E., Cameron, B., and Selva, D., *System architecture: strategy and product development for complex systems*, Pearson Education, England, 2015. https://doi.org/10.1007/978-1-4020-4399-4.

[2] Ulrich, K., "The role of product architecture in the manufacturing firm," *Research Policy*, Vol. 24, No. 3, 1995, pp. 419–440. https://doi.org/10.1016/0048-7333(94)00775-3.

[3] La Rocca, G., "Knowledge Based Engineering Techniques to Support Aircraft Design and Optimization," Ph.D. thesis, Delft University of Technology, 2011.

[4] Roelofs, M., and Vos, R., "Correction: Uncertainty-Based Design Optimization and Technology Evaluation: A Review," *2018 AIAA Aerospace Sciences Meeting*, Reston, Virginia, 2018. https://doi.org/10.2514/6.2018-2029.c1.

[5] McDermott, T., Folds, D., and Hallo, L., "Addressing Cognitive Bias in Systems Engineering Teams," *30th Annual INCOSE International Symposium*, Virtual Event, 2020. https://doi.org/10.1002/j.2334-5837.2020.00721.x.

[6] Judt, D., and Lawson, C., "Development of an automated aircraft subsystem architecture generation and analysis tool," *Engineering Computations*, Vol. 33, No. 5, 2016, pp. 1327–1352. https://doi.org/10.1108/EC-02-2014-0033.

[7] Bussemaker, J. H., and Ciampa, P., "MBSE in Architecture Design Space Exploration," *Handbook of Model-Based Systems Engineering*, edited by A. Madni, N. Augustine, and M. Sievers, Springer, Switzerland, 2022. https://doi.org/10.1007/978-3-030-27486-3_36-1.

[8] Bussemaker, J. H., Bartoli, N., Lefebvre, T., Ciampa, P. D., and Nagel, B., "Effectiveness of Surrogate-Based Optimization Algorithms for System Architecture Optimization," *AIAA AVIATION 2021 FORUM*, Virtual Event, 2021. https://doi.org/10.2514/6.2021-3095.

[9] Bussemaker, J. H., Ciampa, P. D., and Nagel, B., "System Architecture Design Space Exploration: An Approach to Modeling and Optimization," *AIAA AVIATION 2020 FORUM*, Virtual Event, 2020. https://doi.org/10.2514/6.2020-3172.

[10] Martins, J. R. R. A., and Ning, A., *Engineering Design Optimization*, Cambridge University Press, Cambridge, 2022. URL https://mdobook.github.io/.

[11] Bussemaker, J. H., Ciampa, P. D., and Nagel, B., "System Architecture Design Space Modeling and Optimization Elements," *32nd Congress of the International Council of the Aeronautical Sciences, ICAS 2020*, Shanghai, China, 2021.

[12] Haberfellner, R., de Weck, O., Fricke, E., and Vössner, S., *Systems Engineering*, Springer International Publishing, Cham, 2019. https://doi.org/10.1007/978-3-030-13431-0.

[13] Madni, A., "Novel Options Generation," *Transdisciplinary Systems Engineering*, Springer International Publishing, Cham, 2017, pp. 89–102. https://doi.org/10.1007/978-3-319-62184-5_6.

[14] Menshenin, Y., Mordecai, Y., Crawley, E. F., and Cameron, B. G., "Model-Based System Architecting and Decision-Making," *Handbook of Model-Based Systems Engineering*, Springer International Publishing, Cham, 2022, pp. 1–42. https://doi.org/10.1007/978-3-030-27486-3_17-1.

[15] Broodney, H., Dotan, D., Greenberg, L., and Masin, M., "1.6.2 Generic Approach for Systems Design Optimization in MBSE1," *INCOSE International Symposium*, Vol. 22, No. 1, 2012, pp. 184–200. https://doi.org/10.1002/j.2334-5837.2012.tb01330.x.

[16] Weilkiens, T., *Variant Modeling with SysML*, Leanpub, Victoria, BC, Canada, 2015.

[17] Bajaj, M., Friedenthal, S., and Seidewitz, E., "Systems Modeling Language (SysML v2) Support for Digital Engineering," *INSIGHT*, Vol. 25, No. 1, 2022, pp. 19–24. https://doi.org/10.1002/inst.12367.

[18] Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., and Wasowski, A., "Cool features and tough decisions," *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems - VaMoS '12*, ACM Press, Leipzig, Germany, 2012. https://doi.org/10.1145/2110147.2110167.

[19] Madeira, R. H., de Sousa Pinto, D. H., and Forlingieri, M., "Variability on System Architecture using Airbus MBPLE for MOFLT Framework," *33rd Annual INCOSE International Symposium*, 2023. https://doi.org/10.1002/iis2.13041.

[20] Gedell, S., and Johannesson, H., "Design rationale and system description aspects in product platform design: Focusing reuse in the design lifecycle phase," *Concurrent Engineering*, Vol. 21, No. 1, 2012, pp. 39–53. https://doi.org/10.1177/1063293x12469216.

[21] Raudberget, D., Edholm, P., and Andersson, M., "Implementing the principles of Set-based Concurrent Engineering in Configurable Component Platforms," *DS 71: Proceedings of NordDesign 2012, the 9th NordDesign conference*, Aarlborg University, Denmark, 2012.

[22] Wyatt, D., Wynn, D., Jarrett, J., and Clarkson, P., "Supporting product architecture design using computational design synthesis with network structure constraints," *Research in Engineering Design*, Vol. 23, No. 1, 2012, pp. 17–52. https://doi.org/10.1007/s00163-011-0112-y.

[23] Kirov, D., Nuzzo, P., Passerone, R., and Sangiovanni-Vincentelli, A., "ArchEx: An Extensible Framework for the Exploration of Cyber-Physical System Architectures," *Proceedings of the 54th Annual Design Automation Conference 2017*, ACM, New York, NY, USA, 2017. https://doi.org/10.1145/3061639.3062204.

[24] Herber, D. R., "Enhancements to the Perfect Matching Approach for Graph Enumeration-Based Engineering Challenges," *Volume 11A: 46th Design Automation Conference (DAC)*, American Society of Mechanical Engineers, 2020. https://doi.org/10.1115/detc2020-22774.

[25] Paparistodimou, G., "Generative design of robust modular system architectures," Ph.D. thesis, 2020. https://doi.org/10.48730/T9JH-6R87.

[26] de Vos, P., Stapersma, D., Duchateau, E., and van Oers, B., "Design space exploration for on-board energy distribution systems: A new case study," *Proceedings of the 17th International Conference on Computer and IT Applications in the Maritime Industries (COMPIT '18)*, Pavone, Italy, 2018.

[27] Menu, J., Nicolai, M., and Zeller, M., "Designing Fail-Safe Architectures for Aircraft Electrical Power Systems," *2018 AIAA/IEEE Electric Aircraft Technologies Symposium*, American Institute of Aeronautics and Astronautics, 2018. https://doi.org/10.2514/6.2018-5032.

[28] Nicolai, M., Salemio, L., and Vanhuyse, J., "Design Space Modeling Language for the Generation of Engineering Designs," , Jan. 2020.

[29] Mavris, D., de Tenorio, C., and Armstrong, M., "Methodology for Aircraft System Architecture Definition," *46th AIAA Aerospace Sciences Meeting and Exhibit*, American Institute of Aeronautics and Astronautics, Reston, Virigina, 2008, pp. 1–14. https://doi.org/10.2514/6.2008-149.

[30] Chakraborty, I., and Mavris, D. N., "Integrated Assessment of Aircraft and Novel Subsystem Architectures in Early Design," *54th AIAA Aerospace Sciences Meeting*, Vol. 54, American Institute of Aeronautics and Astronautics, Reston, Virginia, 2016, pp. 1268–1282. https://doi.org/10.2514/6.2016-0215.

[31] Guerster, M., and Crawley, E., "Dominant Suborbital Space Tourism Architectures," *Journal of Spacecraft and Rockets*, Vol. 34385, 2019, pp. 1–13. https://doi.org/10.2514/1.A34385.

[32] Kurtoglu, T., and Campbell, M. I., "Automated synthesis of electromechanical design configurations from empirical analysis of function to form mapping," *Journal of Engineering Design*, Vol. 20, No. 1, 2009, pp. 83–104. https://doi.org/10.1080/09544820701546165.

[33] Albarello, N., Welcomme, J., and Reyterou, C., "A formal design synthesis and optimization method for systems architectures," *Proceedings of MOSIM*, Bordeaux, France, 2012.

[34] Bornholdt, R., Kreitz, T., and Thielecke, F., "Function-Driven Design and Evaluation of Innovative Flight Controls and Power System Architectures," *SAE Technical Paper Series*, SAE International, 2015. https://doi.org/10.4271/2015-01-2482.

[35] Shougarian, N., "Towards concept generation and performance-complexity tradespace exploration of engineering systems using convex hulls," Ph.D. thesis, MIT, Department of Aeronautics and Astronautics, 2017.

[36] Simmons, W., "A Framework for Decision Support in Systems Architecting," Ph.D. thesis, Massachusetts Institute of Technology, 2008.

[37] Iacobucci, J., "Rapid Architecture Alternative Modeling (Raam): a Framework for Capability-Based Analysis of System of Systems Architectures," Ph.D. thesis, Georgia Institute of Technology, 2012.

[38] Franzén, L. K., Staack, I., Krus, P., Jouannet, C., and Amadori, K., "Ontology-Represented Design Space Processing," *AIAA AVIATION 2021 FORUM*, American Institute of Aeronautics and Astronautics, 2021. https://doi.org/10.2514/6.2021-2426.

[39] Apaza, G., and Selva, D., "Automatic Composition of Encoding Scheme and Search Operators in System Architecture Optimization," *41st Computers and Information in Engineering Conference (CIE)*, American Society of Mechanical Engineers, Virtual, 2021. https://doi.org/10.1115/detc2021-71399.

[40] Selva, D., "Rule-based system architecting of Earth observation satellite systems," Ph.D. thesis, Massachusetts Institute of Technology, Dept. of Aeronautics and Astronautics, 2012.

[41] Lopez-Herrejon, R. E., Linsbauer, L., and Egyed, A., "A systematic mapping study of search-based software engineering for software product lines," *Information and Software Technology*, Vol. 61, 2015, pp. 33–51. https://doi.org/10.1016/j.infsof.2015.01.008.

[42] Lazreg, S., Cordy, M., Collet, P., Heymans, P., and Mosser, S., "Multifaceted Automated Analyses for Variability-Intensive Embedded Systems," *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019. https://doi.org/10.1109/icse.2019.00092.

[43] Ölvander, J., Lundén, B., and Gavel, H., "A computerized optimization framework for the morphological matrix applied to aircraft conceptual design," *Computer-Aided Design*, Vol. 41, No. 3, 2009, pp. 187–196. https://doi.org/10.1016/j.cad.2008.06.005.

[44] Frank, C., "A Design Space Exploration Methodology to Support Decisions under Evolving Requirements Uncertainty and its Application to Suborbital Vehicles," Ph.D. thesis, Georgia Institute of Technology, 2016. https://doi.org/10.2514/6.2015-1010.

[45] Selva, D., Cameron, B., and Crawley, E., "Patterns in System Architecture Decisions," *Systems Engineering*, Vol. 19, No. 6, 2016, pp. 477–497. https://doi.org/10.1002/sys.21370.

[46] Selva, D., Cameron, B., and Crawley, E. F., "A rule-based method for scalable and traceable evaluation of system architectures," *Research in Engineering Design*, Vol. 25, No. 4, 2014, pp. 325–349. https://doi.org/10.1007/s00163-014-0180-x.

[47] Frank, C., Marlier, R., Pinon-Fischer, O., and Mavris, D., "An Evolutionary Multi-Architecture Multi-Objective Optimization Algorithm for Design Space Exploration," *57th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, Reston, Virginia, 2016, pp. 1–19. https://doi.org/10.2514/6.2016-0414.

[48] Bussemaker, J. H., Saves, P., Bartoli, N., Lefebvre, T., Lafage, R., and Nagel, B., "System Architecture Optimization Strategies: Dealing with Expensive Hierarchical Problems," *Journal of Global Optimization*, 2024. Article submitted.

[49] Bussemaker, J. H., Saves, P., Bartoli, N., Lefebvre, T., and Nagel, B., "Surrogate-Based Optimization of System Architectures Subject to Hidden Constraints," *AIAA AVIATION 2024 FORUM*, Las Vegas, NV, USA, 2024.

[50] Chaudemar, J.-C., and de Saqui-Sannes, P., "MBSE and MDAO for Early Validation of Design Decisions: a Bibliography Survey," IEEE, 2021. https://doi.org/10.1109/syscon48628.2021.9447140.

[51] Dori, D., "Developing Industry 4 Systems with OPM ISO 19450 Augmented with MAXIM," *Handbook of Model-Based Systems Engineering*, Springer International Publishing, Cham, 2022, pp. 1–20. https://doi.org/10.1007/978-3-030-27486-3_38-1.

[52] Paredis, C., Bernard, Y., Burkhart, R., de Koning, H., Friedenthal, S., Fritzson, P., Rouquette, N., and Schamai, W., "An overview of the SysML-Modelica transformation specification," *INCOSE International Symposium*, Vol. 2, 2010, pp. 1–14. https://doi.org/10.1002/j.2334-5837.2010.tb01099.x.

[53] Bile, Y., Riaz, A., Guenov, M., and Molina-Cristobal, A., "Towards Automating the Sizing Process in Conceptual (Airframe) Systems Architecting," *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, , No. January, 2018, pp. 1–24. https://doi.org/10.2514/6.2018-1067.

[54] Sobieszczanski-Sobieski, J., Morris, A., and van Tooren, M., *Multidisciplinary Design Optimization Supported by Knowledge Based Engineering*, John Wiley & Sons, Ltd, West Sussex, UK, 2015. https://doi.org/10.1002/9781118897072.

[55] Helle, P., Schramm, G., Klostermann, S., and Feo-Arenis, S., "Enabling Multidisciplinary-Analysis of SysML Models in a Heterogeneous Tool Landscape using Parametric Analysis Models," *The Complex Systems Design & Management Conference (CSD&M 2022)*, 2022.

[56] Bussemaker, J. H., Boggero, L., and Ciampa, P. D., "From System Architecting to System Design and Optimization: A Link Between MBSE and MDAO," *32nd Annual INCOSE International Symposium*, Detroit, MI, USA, 2022. https://doi.org/10.1002/iis2.12935.

[57] Bruggeman, A.-L., Nikitin, M., La Rocca, G., and Bergsma, O., "Model-Based Approach for the Simultaneous Design of Airframe Components and their Production Process Using Dynamic MDAO Workflows," *AIAA SCITECH 2024 Forum*, American Institute of Aeronautics and Astronautics, 2024. https://doi.org/10.2514/6.2024-1530.

[58] Sonneveld, J., van den Berg, T., La Rocca, G., Valencia-Ibáñez, S., van Manen, B., and Bruggeman, A., "Dynamic workflow generation applied to aircraft moveable architecture optimization," 2023. https://doi.org/10.13009/EUCASS2023-544.

[59] Bruggeman, A. M., and La Rocca, G., "From Requirements to Product: an MBSE Approach for the Digitalization of the Aircraft Design Process," *INCOSE International Symposium*, Vol. 33, No. 1, 2023, pp. 1688–1706. https://doi.org/10.1002/iis2.13107.

[60] Bussemaker, J. H., De Smedt, T., La Rocca, G., Ciampa, P. D., and Nagel, B., "System Architecture Optimization: An Open Source Multidisciplinary Aircraft Jet Engine Architecting Problem," *AIAA AVIATION 2021 FORUM*, Virtual Event, 2021. https://doi.org/10.2514/6.2021-3078.

[61] Fouda, M., Adler, E. J., Bussemaker, J. H., Martins, J. R. R. A., Kurtulus, D. F., Boggero, L., and Nagel, B., "Automated Hybrid Propulsion Model Construction for Conceptual Aircraft Design and Optimization," *33rd Congress of the International Council of the Aeronautical Sciences, ICAS 2022*, Stockholm, Sweden, 2022.

[62] Bussemaker, J. H., García Sánchez, R., Fouda, M., Boggero, L., and Nagel, B., "Function-Based Architecture Optimization: An Application to Hybrid-Electric Propulsion Systems," *33rd Annual INCOSE International Symposium*, Honolulu, HI, USA, 2023. https://doi.org/10.1002/iis2.13020.

[63] Garg, S., García Sánchez, R., Bussemaker, J. H., Boggero, L., and Nagel, B., "Dynamic Formulation and Excecution of MDAO Workflows for Architecture Optimization," *AIAA AVIATION 2024 FORUM*, Las Vegas, NV, USA, 2024.

[64] Jeyaraj, A., Bussemaker, J. H., Liscouët-Hanke, S., and Boggero, L., "Systems Architecting: A Practical Example of Design Space Modeling and Safety-Based Filtering within the AGILE4.0 Project," *33rd Congress of the International Council of the Aeronautical Sciences, ICAS 2022*, Stockholm, Sweden, 2022.

[65] Weinberger, E., "Correlated and uncorrelated fitness landscapes and how to tell the difference," *Biological Cybernetics*, Vol. 63, No. 5, 1990, pp. 325–336. https://doi.org/10.1007/bf00202749.

[66] Bussemaker, J. H., "SBArchOpt: Surrogate-Based Architecture Optimization," *Journal of Open Source Software*, Vol. 8, No. 89, 2023, p. 5564. https://doi.org/10.21105/joss.05564.

[67] Bussemaker, J. H., Ciampa, P. D., Singh, J., Fioriti, M., Cabaleiro, C., Wang, Z., Peeters, D., Hansmann, P., Vecchia, P. D., and Mandorino, M., "Collaborative Design of a Business Jet Family Using the AGILE 4.0 MBSE Environment," *AIAA AVIATION 2022 FORUM*, Chicago, USA, 2022. https://doi.org/10.2514/6.2022-3934.

[68] Bartoli, N., Lefebvre, T., Dubreuil, S., Olivanti, R., Priem, R., Bons, N., Martins, J., and Morlier, J., "Adaptive modeling strategy for constrained global optimization with application to aerodynamic wing design," *Aerospace Science and Technology*, Vol. 90, 2019, pp. 85–102. https://doi.org/10.1016/j.ast.2019.03.041.

[69] Bussemaker, J. H., and Firchau, T., "System Architecture Optimization: An Example Application to Space Mission Planning," *MBSE2024 Workshop*, Bremen, Germany, 2024.

[70] Cabaleiro de la Hoz, C., Fioriti, M., and Boggero, L., "Automated generation of aircraft on-board system architectures and filtering through certification specification requirements," *Journal of Physics: Conference Series*, Vol. 2716, No. 1, 2024, p. 012044. https://doi.org/10.1088/1742-6596/2716/1/012044.

[71] Cabaleiro de la Hoz, C., Fioriti, M., Ramm, J., Boggero, L., and Nagel, B., "Automated Evaluation of Performance, Certification and Maintenace Aspects of Aircraft On-board System Architectures During Preliminary Design Stages," *AIAA AVIATION 2024 FORUM*, Las Vegas, NV, USA, 2024.

[72] García Sánchez, R., "Adaptation of an MDO Platform for System Architecture Optimization," mathesis, Delft University of Technology, Delft, NL, Jan. 2024.

[73] Hendricks, E., and Gray, J., "pyCycle: A Tool for Efficient Optimization of Gas Turbine Engine Cycles," *Aerospace*, Vol. 6, No. 8, 2019, p. 87. https://doi.org/10.3390/aerospace6080087.

[74] Gray, J., Hwang, J., Martins, J., Moore, K., and Naylor, B., "OpenMDAO: an open-source framework for multidisciplinary design, analysis, and optimization," *Structural and Multidisciplinary Optimization*, Vol. 59, No. 4, 2019, pp. 1075–1104. https://doi.org/10.1007/s00158-019-02211-z.