System Architecture Optimization Strategies: Dealing with Expensive Hierarchical Problems

Jasper H Bussemaker, Paul Saves, Nathalie Bartoli, Thierry Lefebvre, Rémi Lafage

System Architecture Optimization Strategies: Dealing with Expensive Hierarchical Problems

Jasper H. Bussemaker[1*], Paul Saves[2], Nathalie Bartoli[2], Thierry Lefebvre[2], Rémi Lafage[2]

[1*]Institute of System Architectures in Aeronautics, German Aerospace Center (DLR), Hein-Saß-Weg 22, Hamburg, 21129, Germany.
[2]DTIS, ONERA, 2 Avenue Edouard Belin, Toulouse, 31000, France.

*Corresponding author(s). E-mail(s): jasper.bussemaker@dlr.de;
Contributing authors: paul.saves@onera.fr; nathalie.bartoli@onera.fr;
thierry.lefebvre@onera.fr; remi.lafage@onera.fr;

Abstract

Choosing the right system architecture for the problem at hand is challenging due to the large design space and high uncertainty in the early stage of the design process. Formulating the architecting process as an optimization problem may mitigate some of these challenges. This work investigates strategies for solving System Architecture Optimization (SAO) problems: expensive, black-box, hierarchical, mixed-discrete, constrained, multi-objective problems that may be subject to hidden constraints. Imputation ratio, correction ratio, correction fraction, and max rate diversity metrics are defined for characterizing hierarchical design spaces. This work considers two classes of optimization algorithms for SAO: Multi-Objective Evolutionary Algorithms (MOEA) such as NSGA-II, and Bayesian Optimization (BO) algorithms. A new Gaussian process kernel is presented that enables modeling hierarchical categorical variables, extending previous work on modeling continuous and integer hierarchical variables. Next, a hierarchical sampling algorithm that uses design space hierarchy to group design vectors by active design variables is developed. Then, it is demonstrated that integrating more hierarchy information in the optimization algorithms yields better optimization results for BO algorithms. Several realistic single-objective and multi-objective test problems are used for investigations. Finally, the BO algorithm is applied to a jet engine architecture optimization problem. This work shows that the developed BO algorithm can effectively solve the problem with one order of magnitude less function evaluations than NSGA-II. The algorithms and problems used in this work are implemented in the open-source Python library SBArchOpt.

Keywords: architecture optimization, Bayesian optimization, hierarchical, multi-objective, mixed-discrete, hidden constraints

Nomenclature

c_h	=	Hidden constraint
$c_{v,l}$	=	Value constraint l
CR	=	Correction ratio (d subscript: discrete IR; c subscript: continuous IR)
CRF	=	Correction fraction
DoE	=	Design of experiments
f_m	=	Objective function m
g_k	=	Inequality constraint k
HV	=	Hypervolume
IR	=	Imputation ratio (d subscript: discrete IR; c subscript: continuous IR)
k_{doe}	=	DoE multiplier
PLS	=	Partial least squares regression
$KPLS$	=	Kriging with PLS
MRD	=	Maximum rate diversity
n_{c_v}	=	Number of value constraints
n_f	=	Number of objectives
n_g	=	Number of inequality constraints
n_{act}	=	Number of active design variables in a design vector
n_{batch}	=	Batch size for infill points
$n_{corr,discr}$	=	Number of correct discrete design vectors
n_{doe}	=	Design of experiments size
n_{infill}	=	Total number of infill points generated
n_{kpls}	=	PLS components for KPLS
$n_{parallel}$	=	Maximum number of parallel evaluations
$n_{valid,discr}$	=	Number of valid discrete design vectors
n_x	=	Number of design variables
n_{x_c}	=	Number of continuous design variables
n_{x_d}	=	Number of discrete design variables
$n_{x,grp}$	=	Number of design variables in a sampling group
N_{fe}	=	Number of function evaluations
N_j	=	Number of options for discrete variable j
RD_j	=	Rate diversity of discrete variable j
$\hat{s}$	=	Uncertainty estimate by a surrogate model
$TSFC$	=	Thrust-specific fuel consumption
w	=	Group weight for sampling
$\boldsymbol{x}$	=	Design vector
x_{act}	=	Active design variables in a design vector
$x_{c,i}$	=	Continuous design variable i
x_{corr}	=	Design vector to be corrected

$x_{d,j}$	=	Discrete design variable j
$x_{valid,discr}$	=	Valid discrete design vectors
$\hat{y}$	=	Function value estimate by a surrogate model
δ_i	=	Activeness function for design variable i

1 Introduction

System architecture is an important aspect of any engineered system. It conceptually defines how the system meets its stakeholder expectations by specifying the components the system consists of, how these components fulfill system functions, and how the components are related to each other (Crawley et al., 2015). Designing a system architecture is anything but trivial, involving requirements analysis, interviewing stakeholders, defining the top-level functions the system should fulfill, identifying appropriate technologies for fulfilling these functions, managing complexity, and working with disciplinary experts to come up with a solution (Chan et al., 2022). This process is challenged by the fact that due to the combinatorial nature of architecture decisions, it is infeasible to enumerate, analyze, and compare over every possible architecture alternatives (Iacobucci, 2012). Additionally, engineering teams might be biased towards known solutions, preventing them to consider the full range of possibilities (McDermott et al., 2020), mainly due to the time it takes to analyze one architecture alternative. *System Architecture Optimization* (SAO) is an emerging field, where these challenges are addressed by combining numerical optimization techniques with quantitative performance evaluation of architecture alternatives to automatically (but not exhaustively) search the design space in search of the best architecture(s) for the problem at hand (Judt and Lawson, 2016). To turn an architecting process into an architecture optimization problem, several conditions must be satisfied (Bussemaker et al., 2021):

- It should be possible to quantitatively evaluate the performance of each architecture so that architecture candidates can be compared to each other objectively, by implementing a computational framework that can handle all possible architectures and captures relevant (physical) phenomena with sufficient details.
- Architectural decisions need to be formally modeled, such that a computer program can reason about them, try out different combinations of decisions, and explore promising architecture candidates automatically.
- Appropriate numerical optimization algorithms must be available to solve the architecture optimization problem efficiently.

This work addresses the last point: numerical optimization algorithms that can handle all challenges of architecture optimization problems. For an overview of quantitative architecture evaluation and architecture design space modeling methods, the reader is referred to (Bussemaker and Ciampa, 2022; Bussemaker et al., 2022).

This paper continues with a more detailed introduction of the challenges posed by architecture optimization in Section 2, and a discussion of appropriate optimization algorithm classes and their implementations in Section 3. Kernels for modeling categorical variables in hierarchical GP models are presented in Section 4. Section 5 develops and investigates new sampling and correction algorithms for hierarchical design spaces,

and Section 6 discusses and compares different levels of hierarchy integration in optimization algorithms. The developed strategies are applied to a jet engine architecture optimization problem in Section 7 and Section 8 concludes this paper.

2 Characteristics of Architecture Optimization Problems

Optimization is the automation of a design task: the goal is to find one or more design vectors $\boldsymbol{x}$, representing specific design points, that minimize (or maximize) one or more objective functions $f(\boldsymbol{x})$, while satisfying design constraints $g(\boldsymbol{x}) \leq 0$ (Martins and Ning, 2022). The design vectors (or design points) $\boldsymbol{x}$ are composed of one value for each design variable x. Note that maximization of an objective function is enabled by negating the objective function value, and therefore we treat minimization as the default in the rest of this work. In the context of System Architecture Optimization (SAO), a design point $\boldsymbol{x}$ represents an architecture instance. Compared to for example the work by Frank et al. (Frank et al., 2016), we treat the terms "architecture instance", "architecture alternative", and "design point" as synonymous meaning one possible design vector in the architecture design space.

2.1 Design Space Characteristics

SAO problems are *mixed-discrete*: both continuous and discrete design variables can be part of the problem formulation (Frank et al., 2016; Saves et al., 2021). Continuous design variables x_c can take any value between some lower bound $\underline{x}_c$ and some upper bound $\overline{x}_c$ (inclusive). Discrete design variables x_d can only take one from a finite set of values, encoded as an index between 0 and $N_j - 1$ (inclusive), with N_j representing the number of possible discrete values for discrete design variable $x_{d,j}$. Discrete variables can either be of *integer*, *ordinal* or of *categorical* type: for integer and ordinal variables the order of the values is relevant, for categorical values no notion of value order exists (Saves et al., 2023). The difference between integer and ordinal variables is that integer values operate between two bounds and the variable can take any integer value between these bounds, whereas ordinal variables can take any value of a specified list of integer values. Examples of continuous design variables are wing sweep and engine bypass ratio, an example of an integer variable is the number of seat rows in an aircraft cabin (e.g. between 20 and 40), an example of an ordinal variable is the number of engines for an aircraft (e.g. 1, 2 or 4; order is relevant), and an example of a categorical variable is aircraft power source (there is no order between kerosene, hydrogen and electricity). In the context of discrete architecture choices, function-to-component allocation and component connection are categorical choices, and component or system instantiations are integer or ordinal choices (Bussemaker and Ciampa, 2022).

SAO design spaces feature strong interaction between decisions, in the sense that some decisions might affect which other decisions remain to be taken and/or which options remain available for other decisions. For example, consider the Apollo mission architecture problem analyzed by Simmons (Simmons, 2008): the decision whether or not a lunar-orbit-rendezvous or earth-orbit-rendezvous maneuver will be performed is only relevant if an architecture with a lunar module is selected. Another example

from the same architecture problem is crew assignment: the lunar module can have 1, 2 or 3 crew members, except if there are only 2 crew members in the command module, in which case the lunar module can only contain 1 or 2 crew members. In the launch vehicle design problem by Pelamatti et al. (Pelamatti et al., 2020), the number of stages is a decision (1 to 3), as well as several decisions such as fuel type and dimensions for each of the stages: it follows that if a one-stage architecture is selected, the decisions regarding the second and third stages do not have to be considered. Machine learning hyperparameters tuning problems also exhibit such interactions between design variables (Feurer and Hutter, 2019). In general, decisions constraining available options of other decisions is a common theme in architecture optimization, for example defined using an incompatibility matrix (Armstrong et al., 2008). These interactions between design variables create a *hierarchical* structure in the design space (Zaefferer and Horn, 2018), with design variables higher up in the hierarchy determining which variables lower in the hierarchy are *active* or *inactive*, and/or which options are available for active design variables. The property of being active or inactive is called *activeness* (Bussemaker et al., 2021), and is defined through the activeness function $\delta_i(\boldsymbol{x})$ (Hutter and Osborne, 2013), which returns a 1 indicating active or 0 indicating inactive for design variable x_i. Variables that may be inactive are denoted as *conditionally active*. In architecture optimization problems often only discrete design variables determine the activeness of other design variables, however in the general case also continuous variables could determine activeness as in the work by Zaefferer & Horn (Zaefferer and Horn, 2018). Hierarchical design spaces are also known as conditional design spaces (Feurer and Hutter, 2019) as design variables can be conditionally active, design spaces with tree-structured dependencies (Bergstra et al., 2011; Jenatton et al., 2017; Apaza and Selva, 2021), and variable-size design spaces (Abdelkhalik, 2012; Talbi, 2024) as inactive design variables can also be seen as artifacts of a locally-reduced size of the design space.

Inactive design variables do not influence the performance of the design and are therefore irrelevant for objective and constraint evaluations. If an optimizer is not aware of this, it can however still generate design vectors with different values for inactive design variables, resulting in the possibility of multiple design vectors representing the same design (i.e. the design vector to design mapping is no longer one-to-one) and therefore the same objective and constraint values, and thereby wasting computational resources or preventing the optimizer from finding the optimum. To mitigate this, each inactive design variable can be assigned a canonical value, for example 0 for discrete variables and mid-bounds for continuous variables (Zaefferer and Horn, 2018). The process of replacing inactive design variables with canonical values is called design vector *imputation* (Zaefferer and Horn, 2018; Levesque et al., 2017), and the resulting design vector is called an imputed or canonical design vector. Restricting option availability of active variables lower in the hierarchy is done using *value constraints*, also known as enumeration constraints in the context of architecture optimization (Selva, 2012) or configuration constraints in product line engineering (Czarnecki et al., 2012; Weilkiens et al., 2015). According to the taxonomy by Le Digabel & Wild (Le Digabel and Wild, 2023) value constraints are Non-quantifiable (it is only know whether they are violated, not by how much), Unrelaxable (a design point with a failed value

constraint does not represent anything that can be evaluated), A-priori (they can be corrected without running an evaluation), and Known (they are derived from the design space definition). The process of ensuring value constraints are satisfied by modifying a design vector is called *correction*.

A result of design variable hierarchy is that it is not possible to compute the number of discrete design options and continuous variables from the design variable definitions alone: for a non-hierarchical design space, the number of valid discrete design points $n_{valid,discr}$ is equal to the Cartesian product of all discrete values $\prod N_j$. For hierarchical design spaces, however, the number of valid discrete design points can be much lower than that. For example, for the suborbital vehicle design problem of Frank et al. (Frank et al., 2016) there are 2.8 million total combinations of discrete variables, however there are only 123 thousand (4.4% of 2.8 million) valid designs. Ying et al. (Ying et al., 2019) report 423 thousand unique neural networks in a Cartesian design space of 510 million design vectors, a difference of a ratio 1206. To highlight such differences, we distinguish between the *declared* discrete design space, given by the aforementioned Cartesian product, and the *valid* discrete design space, given by the number of valid discrete design vectors $n_{valid,discr}$. Correction combined with imputation moves a design point from the declared design space to the valid design space: correction ensures a design point is made *correct*, imputation ensures that correct *non-canonical* design points are made canonical and thereby *valid* (correct and canonical). The principles of correction and imputation are visualized in Figure 1. A hypothetical source-to-target assignment problem is shown where 1 or 2 energy consumers are assigned to 1 or 2 energy sources. Two hierarchical interactions arise: if there is only one source, then both consumers have to be assigned to that source (a value constraint), and if there is only one consumer, the source choice for the second consumer is not active (hierarchical activation). To move from the declared (Figure 1 step 1) to valid design space, first design vectors are corrected (step 2) and then inactive design variables are imputed (step 3). The valid design space (step 4) contains the unique remaining design vectors. For this example, the declared design space (step 1) consists of 16 design vectors, whereas the valid design space (step 4) only consists of 8 design vectors, demonstrating this discrepancy typical for hierarchical optimization problems. The combined operations of correction and imputation can be implemented in an optimization problem using a repair operator (Salcedo-Sanz, 2009; Selva, 2012). Table 3 provides an overview of relevant design point statuses in the context of SAO.

2.1.1 Imputation and Correction Ratio

Due to hierarchy, the valid design space might be much smaller than the declared design space. Since this might pose a challenge to optimization algorithms we now introduce a metric to quantify this discrepancy: the ratio between the declared and valid (see Table 3) discrete design space sizes is defined as the discrete *imputation ratio* IR_d:

$$IR_d = \frac{\prod_{j=1}^{n_{x_d}} N_j}{n_{valid,discr}} \tag{1}$$

where $n_{valid,discr}$ is the number of valid discrete design vectors, and N_j is the number of options for discrete variable j. An imputation ratio of 1 indicates a non-hierarchical

	Design variable	Values	Encoded
x_0	Nr of sources S	$\{1, 2\}$	$\{0, 1\}$
x_1	Nr of consumers C	$\{1, 2\}$	$\{0, 1\}$
x_2	Choice of S for C_0	$\{S_0, S_1\}$	$\{0, 1\}$
x_3	Choice of S for C_1	$\{S_0, S_1\}$	$\{0, 1\}$

Value constraints and hierarchical activation:

If there is only one S, all C are connected to S_0:
if $x_0 = 0 \rightarrow x_2 = 0$ and $x_3 = 0$

If there is only one C, choosing S for C_1 is irrelevant:
if $x_1 = 0 \rightarrow \delta_3 = 0$ (x_3 is inactive)

① Cartesian product of all x values

"declared design space"

i_{dv}	x_0	x_1	x_2	x_3
1	0	0	0	0
2	0	0	0	1
3	0	0	1	0
4	0	0	1	1
5	0	1	0	0
6	0	1	0	1
7	0	1	1	0
8	0	1	1	1
9	1	0	0	0
10	1	0	0	1
11	1	0	1	0
12	1	0	1	1
13	1	1	0	0
14	1	1	0	1
15	1	1	1	0
16	1	1	1	1

② Apply constraints & activation

"correct design space"

i_{dv}	x_0	x_1	x_2	x_3
1	0	0	0	
2	0	0	0	
3	0	0	0	
4	0	0	0	
5	0	1	0	0
6	0	1	0	0
7	0	1	0	0
8	0	1	0	0
9	1	0	0	
10	1	0	0	
11	1	0	1	
12	1	0	1	
13	1	1	0	0
14	1	1	0	1
15	1	1	1	0
16	1	1	1	1

③ Impute inactive variables

"imputed design space"

i_{dv}	x_0	x_1	x_2	x_3
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	1	0	0
6	0	1	0	0
7	0	1	0	0
8	0	1	0	0
9	1	0	0	0
10	1	0	0	0
11	1	0	1	0
12	1	0	1	0
13	1	1	0	0
14	1	1	0	1
15	1	1	1	0
16	1	1	1	1

④ All unique valid design vectors

"valid design space"

i_{dv}	x_0	x_1	x_2	x_3
1	0	0	0	0
2	0	1	0	0
3	1	0	0	0
4	1	0	1	0
5	1	1	0	0
6	1	1	0	1
7	1	1	1	0
8	1	1	1	1

corrected

inactive

Fig. 1: Illustration of correction and imputation in hierarchical design spaces, showing how the Cartesian product of all discrete values relates to the set of correct, imputed, and valid design vectors. Correction (step 2) modifies design variables such that value constraints are satisfied. Imputation (step 3) assigns canonical values to inactive design variables.

problem, values higher than 1 indicate hierarchy: for the suborbital vehicle design problem in (Frank et al., 2016), the imputation ratio is $2.8e6/123e3 = 22.8$. The higher the value, the more invalid (non-canonical and non-corrected) vectors would be generated in a random search, and therefore the more difficult it is for an optimization algorithm to search the design space if this effect is not considered. Variability factor (Benavides et al., 2010) as used in software product line engineering is the reciprocal of discrete imputation ratio. The continuous imputation ratio IR_c is defined as follows:

$$IR_c = \frac{n_{valid,discr} \cdot n_{x_c}}{\sum_{l=1}^{n_{valid,discr}} \sum_{i=1}^{n_{x_c}} \delta_i(\boldsymbol{x}_{d,l})} \tag{2}$$

where $\delta_i(\boldsymbol{x}_{d,l})$ is the activeness function for continuous variable i for valid discrete design vector $x_{d,l}$, $n_{valid,discr}$ the number of valid discrete design vectors, and n_{x_c} the number of continuous design variables. A value of 1 indicates that all continuous variables are always active. A higher value indicates that for some discrete design vectors one or more continuous variables are inactive. Note that this formulation assumes that only discrete design variables determine activeness of continuous design variables. The overall imputation ratio for a given optimization problem is given by

the product of the two:

$$IR = IR_d \cdot IR_c \tag{3}$$

To give an example, the simple engine architecture benchmark problem from (Bussemaker et al., 2021) has $n_{valid,discr} = 70$ valid discrete vectors, however the Cartesian product of its discrete design variables indicates 216 declared discrete vectors, resulting in a discrete imputation ratio of $IR_d = 216/70 = 3.09$. The problem also contains 9 continuous design variables, of which 7.14 are active on average as seen over all discrete design vectors, which result in a continuous imputation ratio of $IR_c = 9/7.14 = 1.26$. The overall imputation ratio therefore is $IR = 3.09 \cdot 1.26 = 3.89$. To compare, the example problem from Figure 1 has an imputation ratio of $IR = IR_d = 16/8 = 2$.

Similarly to the discrepancy between the declared and valid design space sizes, we can also quantify the discrepancy between the declared and correct (see Table 3) design spaces sizes. This can help determine how much of the design space hierarchy is due to value constraints that need correction, as opposed to design variable activeness. We define the *correction ratio* CR as:

$$CR_d = \frac{\prod_{j=1}^{n_{x_d}} N_j}{n_{correct,discr}} \tag{4}$$

$$CR_c = \frac{n_{correct,discr} \cdot n_{x_c}}{\sum_{l=1}^{n_{correct,discr}} \sum_{i=1}^{n_{x_c}} \delta_i(\boldsymbol{x}_{d,l})} \tag{5}$$

$$CR = CR_d \cdot CR_c \tag{6}$$

where CR_d is the discrete correction ratio, $n_{correct,discr}$ the number of correct discrete design vectors, and CR_c the continuous correction ratio. The impact of the need of correction to the design space hierarchy can be quantified by the *correction fraction* CRF:

$$CRF = \frac{\log CR}{\log IR} \tag{7}$$

CRF varies between 0% and 100%, where 0% indicates no hierarchy is due to correction ($CR = 1$) and 100% indicates all hierarchy is due to correction ($CR = IR$, and there is no hierarchy due to activeness). The valid design vectors shown in Table 1 represent a design space with a declared size of 12 ($N_0 \cdot N_1 = 4 \cdot 3 = 12$), however $n_{valid,discr} = 6$, and therefore $IR = IR_d = 12/6 = 2.0$. Additionally, the example has $n_{corr,discr} = 10$, because the inactive design variables can take any declared value and still represent a correct (but not necessarily valid) design vector. Therefore, $CR = CR_d = 12/10 = 1.2$. From this, $CRF = \log 1.2 / \log 2.0 = 26\%$, indicating that 26% of the design space hierarchy is due to the need for correction, while the other 74% are due to activeness and the need for imputation.

2.1.2 Rate Diversity

In addition to the potentially large gap between declared and valid design space sizes, there might also be a discrepancy in how often individual discrete values appear in all possible discrete design vectors, as also noted by Crawley et al. (Crawley et al., 2015). They mention that for a partitioning problem of 10 elements, where the goal is to choose the best subset of elements from the available set, there are 115 thousand

Table 1: Example set of valid design vectors, showing inactive variables in red.

i_{dv}	x_0	x_1
1	0	0
2	0	1
3	1	0
4	1	2
5	2	
6	3	

possibilities to choose from, however 88% of those possibilities are made up of the size-4, -5, and -6 subsets. This means that all other sizes are represented much less in the total number of possibilities. This observation can be extended to design variable values too: for the realistic engine architecting benchmark problem of Bussemaker et al. (Bussemaker et al., 2021), there are a little over 142 thousand valid architectures, however only 27 of those (about 0.02%) represent a pure jet engine architecture; the rest are turbofan architectures. Therefore there is a large gap between how often the two possible values of this design variable appear in all discrete design vectors. This gap can be quantified by the *rate diversity* RD_j, which is defined for each discrete design variable $x_{d,j}$, and the *max rate diversity* MRD, which is defined on the problem level:

$$Rates_j = \{Rate(j, \delta_j = 0), Rate(j, 0), .., Rate(j, N_j - 1)\} \tag{8}$$

$$RD_j = \max Rates_j - \min Rates_j \tag{9}$$

$$MRD = \max_{j \in 1,..,n_{x_d}} RD_j \tag{10}$$

with j the index of the discrete design variable, $\delta_j = 0$ denoting the case when design variable j is inactive, and $Rate(j, value)$ returning the relative occurrence rate of that value in all discrete design vectors. Rate diversity RD_j then represents the difference between the most and least occurring values for a given discrete design variable j, and max rate diversity MRD the maximum of all rate diversities. Rate diversity and max rate diversity are normally defined without the inactive case $Rate(j, \delta_j = 0)$ included. If the inactive case is included, the subscript *all* is added to denote all cases and values are considered. Table 2 shows occurrence rates, the rate diversity and maximum rate diversities of the simple turbofan benchmark problem from (Bussemaker et al., 2021). The rate diversity of the choice between pure jet and turbofan architectures x_0 is not as extreme as for the aforementioned realistic benchmark, however there is still a rate diversity of 60% as only 20% of possible discrete design vectors represent a jet engine architecture. The example problem from Figure 1 has two variables (x_0 and x_3) where

Table 2: Rate diversity RD of the simple turbofan architecting problem from (Bussemaker et al., 2021). The maximum rate diversity MRD values are underlined.

	x_1	x_4	x_{11}	x_{13}	x_{14}	x_{15}
Inactive	–	–	20%	20%	7.1%	7.1%
$x_j = 0$	20%	7.1%	40%	40%	35.7%	35.7%
$x_j = 1$	80%	28.6%	40%	40%	35.7%	35.7%
$x_j = 2$	–	64.3%	–	–	21.4%	21.4%
RD_{all}	<u>60%</u>	57.1%	20%	20%	28.6%	28.6%
RD	<u>60%</u>	57.1%	0%	0%	15.4%	15.4%

one value occurs 2 of 8 times (25%) and the other therefore 75%, which results in an max rate diversity of $MRD = 50\%$.

Imputation ratio IR (Equation 3), correction ratio CR (Equation 6) and max rate diversity MRD (Equation 10) can be useful for quantifying how large the impact of hierarchy is for a given problem.

2.2 Solution Space Characteristics

The objective and design constraint functions $f(\boldsymbol{x})$ and $g(\boldsymbol{x})$, also known as the *evaluation functions*, are *black-box* functions: their behavior is not known in advance because the architecture optimization method can be applied to design any system. As a consequence, the evaluation functions exhibit non-linear, non-smooth (the function may contain discontinuities or gaps) and multi-modal behaviors (there may be multiple local minima; this also implies that the function is non-convex), and gradients cannot be assumed to be available (also due to the mixed-discrete nature of the problem, see below) (Crawley et al., 2015). Evaluation functions are assumed to be deterministic: repeated function calls with the same inputs yield the same outputs. In many cases the evaluation performed to obtain the objective and design constraint values is *expensive* in terms of time and/or computational resources, due to the use of physics-based simulations and Multidisciplinary Design Analysis and Optimization (MDAO (Sobieszczanski-Sobieski et al., 2015)) approaches. The consequence of this is that the time to perform one evaluation can be orders of magnitude more than the time for one iteration of the optimization algorithm, and therefore the focus should be on reducing the amount of evaluations, rather than speeding up the optimization algorithm itself (Jones et al., 1998).

When designing a system architecture, needs and goals of system stakeholders often conflict with each other (Crawley et al., 2015) and the associated architecting problem thus becomes a trade-off between these conflicting goals. In general, therefore, the SAO problem should be considered to be a *multi-objective* optimization problem (Judt and Lawson, 2016): the problem will have multiple conflicting objective functions f to minimize at the same time. Due to this conflicting nature, however, it becomes impossible to define one design point as optimal; rather, multi-objective optimization results in a set of Pareto-optimal design points (Miettinen, 1998). This so-called Pareto-set is comprised of design points that are not dominating each other: a design point $\boldsymbol{x}_a$

dominates a design point $\boldsymbol{x}_b$ if $f_m(\boldsymbol{x}_a) \leq f_m(\boldsymbol{x}_b)$ for all m and $f_m(\boldsymbol{x}_a) < f_m(\boldsymbol{x}_b)$ for at least one m. The consequence of this is that within the Pareto-set, no design point is better than any other design point, and that if you want to improve one or more of the objective values by moving from one point to another, you will also always make at least one other objective value worse. The Pareto-front contains the objective values of the points in the Pareto-set. The challenge that multi-objective optimization algorithms face is to simultaneously progress towards the Pareto-front and maintain sufficient diversity along the Pareto-front (Rudenko and Schoenauer, 2004).

Design constraints $g(\boldsymbol{x})$ can arise from physical (e.g. maximum material stresses or temperatures) or operational limitations (e.g. maximum take-off field length, maximum wing span), for example. Solving constrained optimization problems means that a design point can only be considered optimal if all the design constraints are satisfied. In this work, we only consider *inequality* constraints, as equality constraints can also be defined as two inequality constraints (Priem et al., 2019) or can be eliminated by design variable substitution. Furthermore, we assume that the design constraints are QRSK in the taxonomy of Le Digabel & Wild (Le Digabel and Wild, 2023): Quantifiable (the degree of feasibility can be quantified), Relaxable (a violated constraint is still meaningful to the optimizer), Simulation-based (to know whether the constraint is satisfied, an evaluation must be run), and Known (the constraint is defined in the problem formulation). Design points where one or more design constraints are violated are called *infeasible*, as opposed to *feasible* if all constraints are satisfied.

Simulations used for evaluating architecture performance can fail, for example due to an unstable system of equations, infeasible underlying physics, or infeasible geometric parameterization (Forrester et al., 2006). Any problem that employs simulation could include a so-called *hidden constraint* (also known as unknown, unspecified, forgotten, virtual, and crash constraints (Müller and Day, 2019; Le Digabel and Wild, 2023)): a constraint that manifests itself through failed evaluations. The objective f and design constraint g function outputs are assigned NaN (Not a Number) values when a hidden constraint is violated. We assume that hidden constraints are deterministic, resulting in the same status when repeatedly evaluating the same design point. Additionally, finding out about if the hidden constraint is violated takes at least as long as completing a successful evaluation takes, if not longer (Müller and Day, 2019). The region of the design space where the hidden constraint is violated is called the *failed* region consisting of *failed* design points, the region where this is not the case is called the *viable* region consisting of *viable* points. The failed region can span a large part of the design space: Krengel & Hepperle report up to 60% for a wing design problem (Krengel and Hepperle, 2022), and for an airfoil design problem Forrester et al. report up to 82% failed points (Forrester et al., 2006).

Table 3: Possible statuses of design points and regions. In order for any of the conditions to be met, conditions and operations above have to be met and performed as well.

Condition	Status if condition is ... met	not met	Performed operations
Any $\boldsymbol{x}$ in Cartesian product of x_d values	Declared		
All value constraints satisfied ($c_{v,l}(\boldsymbol{x}) = 0$)	Correct	Invalid	Correction
$\boldsymbol{x}$ is canonical	Valid	Non-canonical	Imputation
Hidden constraint satisfied ($c_h(\boldsymbol{x}) = 0$)	Viable	Failed	Evaluation
All design constraint satisfied ($g_k(\boldsymbol{x}) \leq 0$)	Feasible	Infeasible	Optimization
Optimality achieved	Optimal	Non-optimal	Optimization

2.3 Problem Formulation

Combining all relevant behavioral aspects, an SAO problem can be formulated as:

$$\begin{array}{lll} \textit{minimize} & f_m(\boldsymbol{x}, \delta(\boldsymbol{x})), & m = 1, 2, \ldots, n_f \\ \textit{w.r.t.} & \underline{x}_{c,i} \leq x_{c,i} \leq \overline{x}_{c,i} & i = 1, 2, \ldots, n_{x_c} \\ & x_{d,j} \in \{0, .., N_j - 1\} & j = 1, 2, \ldots, n_{x_d} \\ \textit{subject to} & g_k(\boldsymbol{x}, \delta(\boldsymbol{x})) \leq 0, & k = 1, 2, \ldots, n_g \\ & c_{v,l}(\boldsymbol{x}) = 0 & l = 1, 2, \ldots, n_{c_v} \\ & c_h(\boldsymbol{x}) = 0 & \end{array} \tag{11}$$

where $f_m(\boldsymbol{x}, \delta(\boldsymbol{x}))$ represents the multiple objective functions to be minimized, $\boldsymbol{x}$ the design vector consisting of n_{x_c} continuous and n_{x_d} discrete design variables, and $\delta(\boldsymbol{x})$ the activeness function representing design space hierarchy. Continuous design variable $x_{c,i}$ bounds are represented by $\underline{x}_{c,i}$ and $\overline{x}_{c,i}$, and N_j is the number of discrete options for the discrete variable $x_{d,j}$. The inequality design constraint k is given by $g_k(\boldsymbol{x}, \delta(\boldsymbol{x}))$. The unrelaxable value and hidden constraints are defined by the functions $c_{v,l}(\boldsymbol{x})$ and $c_h(\boldsymbol{x})$, respectively, both returning 1 if the constraint is violated and 0 otherwise. Considering all types of constraints, the design points and regions can have several different non-exclusive statuses as explain in the previous sections and summarized in Table 3.

3 Algorithms for System Architecture Optimization

This section discusses optimization algorithms for solving SAO problems. First, appropriate classes of algorithms are discussed in Section 3.1: Multi-Objective Evolutionary Algorithms (MOEA) and Bayesian Optimization (BO) algorithms. Section 3.2 then provides details of the BO algorithm used in this work not related to design space hierarchy. The implementation of optimization algorithms in the SBArchOpt library is discussed in Section 3.3.

3.1 Appropriate Algorithm Classes

The properties of SAO problems discussed in Section 2 have several consequences, also summarized in Table 4, on the types of optimization algorithms that can be used

Table 4: Behavior of architecture optimization problems and needed algorithm capabilities.

Space	Property	Capability needs
Design space	Mixed-discrete	Gradient-free optimization (Martins and Ning, 2022)
	Hierarchical	Correction and imputation (Zaefferer and Horn, 2018)
Solution space	Black-box	Global, gradient-free optimization (Martins and Ning, 2022)
	Expensive	Minimize number of function evaluations (Jones et al., 1998)
	Multi-objective	Find the Pareto-set (Miettinen, 1998)
	Design constraints	Constraint handling (Martins and Ning, 2022)
	Hidden constraints	Failed area avoidance (Müller and Day, 2019)

to solve these problems. The fact that the objective and design constraint functions exhibit non-linear, non-smooth, and multi-modal behavior without gradient availability necessitates using a global, gradient-free optimization algorithm (Martins and Ning, 2022). Gradient-based optimization is additionally excluded from consideration due to the mixed-discrete, hierarchical nature of the design space. Hierarchy induces the need to move design points from the declared to the valid design space by applying correction and imputation (Zaefferer and Horn, 2018). The expensive nature of the evaluation functions drives the need to minimize the number of evaluations needed to find the optimum Pareto-set (Jones et al., 1998). Finally, constraint handling is needed for both the design constraints (Martins and Ning, 2022) and hidden constraints (Müller and Day, 2019).

Many classes of global optimization algorithms exist, designed for various purposes (Martins and Ning, 2022). Exact optimization methods such as grid search (Yang and Shami, 2020) or DIRECT (Jones and Martins, 2020) are designed to yield reproducible results with provable convergence behavior, and are relevant for problems where finding the true optimum has a high priority (Locatelli and Schoen, 2021), however exact methods struggle to solve high-dimensional problems (Jones and Martins, 2020). On the other hand, heuristic optimization methods depend on strategies that work well in practice to search a design space more efficiently than exact optimization methods, without providing mathematical proof of convergence (Glover and Kochenberger, 2003). One of the most powerful classes of heuristic global optimization methods are *evolutionary algorithms* (EA). Evolutionary algorithms are population-based algorithms that mimic evolutionary processes found in nature: an initial population of individuals (design points) is evolved (optimized) for maximum fitness (objective value) by generating offspring (new design points) from selected parents whose genes (portions of the design vector) are crossed-over and mutated (Petrowski and Ben-Hamida, 2017). Major variations between evolutionary algorithms lie in the way design points are encoded (i.e. the encoding grammar (Selva, 2012)), how parents are selected for the basis of the new generation, types of cross-over and mutation operators used, and how the new generation is created from the current generation and offspring by a survival operator. Evolutionary algorithms are robust in dealing with mixed-discrete design spaces: dealing with continuous variables was in fact a development that came later, for example through Differential Evolution (Petrowski and

Table 5: Past applications of evolutionary algorithms to architecture optimization problems. Nomenclature: number of discrete design points $n_{valid,discr}$, number of continuous dimensions n_{x_c}, number of objectives n_f, number of function evaluations N_{fe}, Ant Colony Optimization (ACO), Genetic Algorithm (GA), Non-dominated Sorting GA II (NSGA-II), Multi-Objective Evolutionary Algorithms (MOEA), Guidance, Navigation and Control (GN&C).

		Design space				
Algorithm(s)	Application	$n_{valid,discr}$	n_{x_c}	n_f	N_{fe}	Reference
GA	Supersonic aircraft	576	100	10	200 000	(Buonanno, 2005)
([1])	Satellite instrument selection	8.8e12		4	20 000	(Selva, 2012)
NSGA-II	Aircraft family	21 875	4	4	200 000	(Pate et al., 2012)
NSGA-II	Suborbital launch vehicle	123 000	23	3	50 000	(Frank et al., 2016)
ACO + GA	Test aircraft subsystem	9 600		2	96 000	(Judt and Lawson, 2016)
NSGA-II	Aircraft engine	1 163	30	3 ([2])	4 000	(Bussemaker et al., 2021)
MOEA	GN&C system	79e6		2	1 000	(Apaza and Selva, 2021)

[1] A special-purpose architecture optimization evolutionary algorithm was used.

[2] The problem also included 15 design constraints.

Ben-Hamida, 2017) or CMA-ES (Hamano et al., 2022). Design constraints are handled either by applying a penalty on the objective functions or by integrating constraints in the selection and survival operators directly (Petrowski and Ben-Hamida, 2017). Multi-Objective Evolutionary Algorithms (MOEA) have been developed to deal with multi-objective problems by modifying selection and survival operators for searching for a Pareto-front rather than a single optimal point (Petrowski and Ben-Hamida, 2017). One of the most popular MOEA is the Non-dominated Sorting Genetic Algorithm II (NSGA-II) (Deb et al., 2002) due to its low configuration effort and good performance. Finally, EA can explore hierarchical design spaces using the hidden genes approach (Abdelkhalik, 2012; Nyew et al., 2015; Talbi, 2024) and deal with hidden constraints using the extreme barrier approach (Gratton and Vicente, 2014). The appropriateness of evolutionary algorithms in general and NSGA-II in particular for SAO is noted by Crawley et al. (Crawley et al., 2015) and has been demonstrated in the past by various applications, see Table 5. Evolutionary algorithms have also found application for optimizing software product lines (Lopez-Herrejon et al., 2015), which involves optimization problems characterized by strong hierarchy and choice dependencies, similarly to architecture optimization.

As powerful as evolutionary algorithms are for dealing with the challenges of architecture optimization, there is one aspect that poses a problem if the objective and constraint functions are expensive to evaluate: the high number of needed function evaluations N_{fe} (Chugh et al., 2019). As can be seen in Table 5, N_{fe} typically is in the order of thousands to hundreds of thousands for EA. Each evaluation of the aircraft engine problem of Bussemaker et al. (Bussemaker et al., 2021) took about two minutes, resulting in a little over 5 days to complete the 4000 evaluations. Depending on the context this could still be acceptable, however the time required to solve an optimization problem driven by an evolutionary algorithm quickly becomes intractable if more evaluations are needed, or if evaluation becomes more costly if

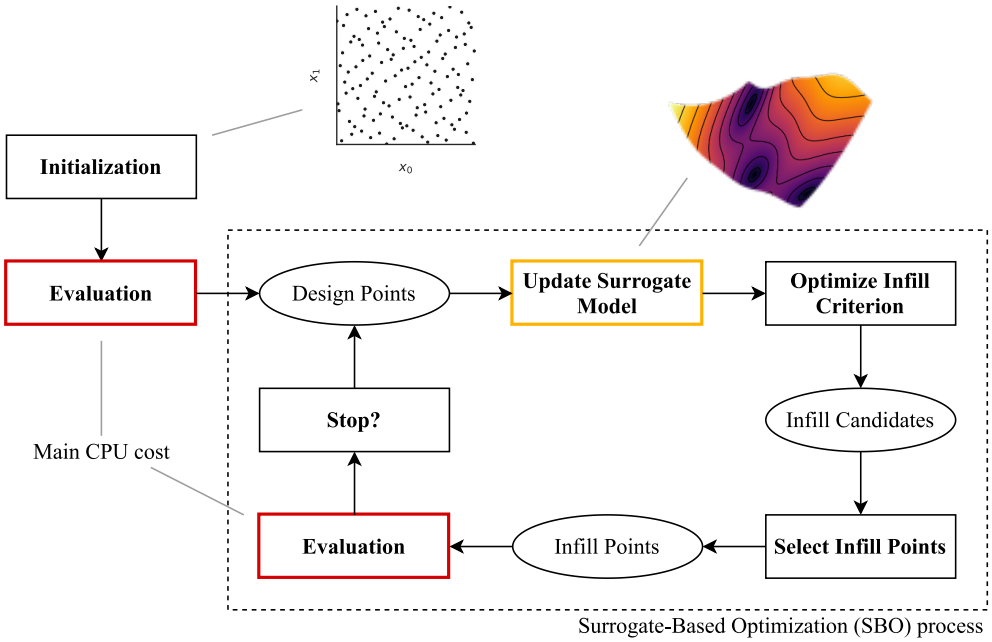


Fig. 2: Principle of Surrogate-Based Optimization (SBO), adopted from (Bussemaker et al., 2021)

more or higher fidelity analyses are used for architecture evaluation. To solve such problems, *Surrogate-based Optimization (SBO)* algorithms have been developed (Martins and Ning, 2022). SBO algorithms rely on some surrogate model, also known as response surface model or metamodel, of the objective and constraint functions in order to efficiently determine interesting design point(s) to evaluate, the *infill* point(s). Once the infill points have been evaluated using the expensive evaluation functions, the surrogate model is reconstructed and the process starts over, until some termination criterion has been reached. Infill points are selected using infill criteria, also known as acquisition functions, that are normally defined to represent some kind of trade-off between exploration (finding new interesting regions) and exploitation (improving already interesting regions) of the design space. Figure 2 visualizes the working principle of SBO.

The main choices involved in using SBO are the initial sampling method for creating the first surrogate model, the type of surrogate model, the infill criterion, and the termination criterion. For example, Regis & Shoemaker (Regis and Shoemaker, 2013) use a Radial Basis Function (RBF) surrogate with a Coordinate Perturbation infill criterion, striking a balance between the best predicted objective value (exploitation) and staying away from previously evaluated points (exploration). SBO has also been extended with strategies for dealing with hidden constraints (Müller and Day, 2019). Especially powerful types of SBO algorithms are *Bayesian Optimization (BO)* algorithms (Garnett, 2023). BO algorithms use surrogate models that provide error

estimates $\hat{s}(\boldsymbol{x})$ in addition to function estimates $\hat{y}(\boldsymbol{x})$, and therefore can use infill criteria that leverage this additional information in order to better predict where interesting points lie. Most applications of BO use Gaussian Process (GP) models (Chugh et al., 2019), also known as Kriging models. Extensive research has been performed on handling design constraints (Schonlau et al., 1998; Sasena, 2002) and solving multi-objective (Knowles, 2006; Rojas-Gonzalez and Van Nieuwenhuyse, 2019) problems using BO. GP models were originally developed for continuous variables, however research has been performed into methods for supporting mixed-discrete (Garrido-Merchán and Hernández-Lobato, 2020; Daulton et al., 2022; Pelamatti et al., 2019; Zuniga and Sinoquet, 2020; Saves et al., 2023; Dreczkowski et al., 2023) and hierarchical (Pelamatti et al., 2020; Audet et al., 2023; Zaefferer and Horn, 2018; Horn et al., 2019; Lu et al., 2018; Jenatton et al., 2017; Saves et al., 2024) design spaces. Enabling the use of GP models for high-dimensional design spaces is an active area of research (Bouhlel et al., 2018; Garnett, 2023; Priem et al., 2023). For the especially non-smooth and hierarchical design spaces encountered in hyperparameter optimization problems Random Forest (RF) (Hutter et al., 2011; Lindauer et al., 2022) and Tree Parzen Estimator (TPE) (Bergstra et al., 2011; Ozaki et al., 2022) models have been applied. These two models naturally represent the tree structure of hierarchical design spaces (Eggensperger et al., 2015), however recent evidence shows the superior performance of mixed-discrete GP models over RF and TPE models (Garrido-Merchán and Hernández-Lobato, 2020), and therefore in this work we will concentrate on the use of BO with GP models only.

In the rest of this work, both Multi-Objective Evolutionary Algorithms (MOEA) and Bayesian Optimization (BO) algorithms will be further investigated for use in architecture optimization problems. One is not better than the other: MOEA should be used if evaluation is not expensive (e.g. one evaluation takes at most in the order of seconds), and BO should be used if evaluation is expensive (e.g. in the order of minutes or more) (Gamot et al., 2023). Bayesian Optimization should not be used for inexpensive problems, as then time for model fitting and searching for infill points becomes limiting, rather than function evaluation time (Chugh et al., 2019).

3.2 Non-hierarchical Foundation of the Bayesian Optimization Algorithm

An advantage of BO is the high level of composability: many specializations of BO (e.g. mixed-discrete, multi-objective) can be combined without negative interactions (Greenhill et al., 2020). In this work we follow the same approach: specializations, e.g. hierarchy and hidden constraints, build on prior specializations, e.g. mixed-discrete and multi-objective. The GP model used by the BO algorithm further developed in this work was already able to deal with mixed-discrete variables, including categorical variables, for non-hierarchical design spaces. Categorical variables pose a particular challenge to GP models, since there is no inherent ordering in the possible values and therefore conventional distance measures used for modeling correlation for continuous and integer variables might not be able to model the variable accurately. The approach for modeling categorical variables is based on the work by Saves et al. (Saves et al., 2023), which uses dedicated kernels to model categorical variables, for example the

Gower or Exponential Homoscedastic Hypersphere (EHH) kernel. We use the implementation in the Surrogate Modeling Toolbox (SMT)[1] (Saves et al., 2024). By default the Gower distance kernel is used.

One of the main limitations of BO is that GP models do not handle high-dimensional design spaces well. Due to the number of hyperparameters to tune, training and sampling time quickly increases with the number of input dimensions and training points (Garnett, 2023). However, a feature space can be defined that reduces the dimension of the original input space (Calandra et al., 2016), supported by the observation that in some cases the optimization problem has a lower intrinsic dimensionality. The BO algorithm uses Kriging with Partial Least Squares (KPLS) to construct such feature spaces as described by Bouhlel et al. (Bouhlel et al., 2016). Recently KPLS has been extended to work with discrete variables too (Saves et al., 2022; Charayron et al., 2023) and these methods are implemented in SMT (Bouhlel et al., 2019; Saves et al., 2024). By default, KPLS is applied with $n_{kpls} = 10$ if the design space contains more than 10 design variables.

Infill points are selected by formulating an infill ensemble: an approach first presented by Lyu et al. (Lyu et al., 2018) based on the observation that different infill criteria can suggest very different infill points. This can be especially true in the earlier phases of the optimization when the GP is less accurate. There are two advantages to this ensemble-infill strategy (Cowen-Rivers et al., 2020): the selected infill points represent the best trade-off between the infill criteria, and it is easy to select multiple infill points per iteration for batch optimization (Garnett, 2023) without needing to retrain the underlying GP models as is needed for qEI (Ginsbourger et al., 2010). For single-objective optimization, an ensemble of Lower Confidence Bound (LCB) (Cox and John, 1992), Expected Improvement (EI) (Jones et al., 1998) and Probability of Improvement (PoI) (Hawe and Sykulski, 2007) infills is used. For multi-objective, the ensemble consists of the Minimum Probability of Improvement (MPoI) (Rahat et al., 2017) and Minimum Euclidean PoI (MEPoI) (Bussemaker et al., 2021) infills. The infill batch size n_{batch} is set to the maximum amount of designs that can be evaluated in parallel: $n_{batch} = n_{parallel}$.

Design constraints are handled by constraint function mean prediction $\hat{g}$ (Sasena, 2002) by default, or by Probability of Feasibility (PoF) (Sohst et al., 2021) if a more or less conservative (achieved by PoF above or below 50%, respectively) criterion is requested. Hidden constraints are handled by training an additional model to predict the Probability of Viability (PoV), and constraining it to be at least 25% (by default) during infill search. The PoV prediction model is the same mixed-discrete GP as used for predicting f and g.

3.3 Implementation in SBArchOpt

The optimization algorithms presented in this work are implemented in the SBArchOpt[2] library: an open-source Python library for solving SAO problems (Bussemaker, 2023). SBArchOpt features a problem definition class with interfaces for executing hierarchical optimization problems. The problem definition class is built on

[1] https://smt.readthedocs.io/
[2] https://sbarchopt.readthedocs.io/

top of PYMOO's[3] PROBLEM class (Blank and Deb, 2020), and adds functions for performing imputation and correction (standalone, not part of design point evaluation), getting information about which design variables are conditionally active, generating all valid discrete design vectors $x_{valid,discr}$, and for getting various statistics (see Section 2.1). The hierarchical structure of the problem can either be provided implicitly by implementing the correction and imputation function, or it can be modeled explicitly using CONFIGSPACE[4] (Lindauer et al., 2019). CONFIGSPACE enables declaring design variables, specifying activation conditions between them, and specifying value constraints. For more discussion about the role CONFIGSPACE can play in hierarchical optimization, refer to Section 6.

SBARCHOPT implements various optimization algorithms for solving architecture optimization problems. These optimization algorithms are implemented using PYMOO classes, enabling provisioning of any PYMOO algorithm for solving architecture optimization problems. For example, SBARCHOPT provides the hierarchical sampling algorithm presented in Section 5 as a PYMOO class. Similarly, a repair operator is implemented that either uses the problem-agnostic correction algorithm of Section 5, or uses the correction function supplied by the problem definition class. Additionally, SBARCHOPT implements result storage and restart functionalities, which are important for expensive and therefore long-running optimization problems. Two pre-configured PYMOO algorithms are provided: ARCHOPTNSGA2 implementing NSGA-II (Deb et al., 2002) for architecture optimization, and DOEALGORITHM that only executes the hierarchical sampling algorithm to perform a Design of Experiments (DoE). The SBO algorithm developed in this work is implemented as ARCHSBO, and features automatic selection of infill and hidden constraints strategies. It can either use an RBF model or a GP model, both using the implementation in the Surrogate Modeling Toolbox (SMT)[1] (Saves et al., 2024). To test and promote solving architecture optimization problems with SBO in general, SBARCHOPT also implements connections to other SBO libraries such as BOTORCH (Balandat et al., 2019), TRIESTE (Picheny et al., 2023), HEBO (Cowen-Rivers et al., 2020), SEGOMOE (Bartoli et al., 2019) and SMARTY (Bekemeyer et al., 2022).

Finally, to support SAO algorithm development SBARCHOPT contains a library of test problems featuring various combinations of continuous or mixed-discrete, (non-)hierarchical, single-objective or multi-objective, (un)constrained problems with or without hidden constraints. Many problems are implemented from literature or adopted from PYMOO's test problem database, however also several new test problems are provided. Realistic engineering problem behavior is exhibited by the jet engine architecture optimization problem used in Section 7, a multi-stage launch vehicle architecture problem adopted from (García Sánchez, 2024) and several version of a Guidance, Navigation & Control (GNC) problem adopted from (Crawley et al., 2015; Apaza and Selva, 2021), see also Section 5.1.

[3] https://pymoo.org/
[4] https://automl.github.io/ConfigSpace/

4 Modeling Hierarchical Design Spaces in Bayesian Optimization

In order to apply SBO to architecture optimization problems, the surrogate model should be able to accurately model the behavior of the objective and constraint functions over the design space. For architecture optimization problems it means that the hierarchical structure between design variables should be integrated in the structure of the surrogate model. Gaussian Process (GP) models as used in Bayesian Optimization (BO) have been extended to support hierarchical design variables by Saves et al. (Saves et al., 2024) and implemented in version 2.0 of the Surrogate Modeling Toolbox (SMT)[1]. This work, however, only considered continuous and integer hierarchical variables. Here we provide an extension to previous work that also supports categorical conditionally active variables. Based on the advances published in (Saves et al., 2023), we can define a new hierarchical kernel for categorical conditional variables based on one-hot encoding and on the algebraic distance defined in (Saves et al., 2024). The new hierarchical kernel between points r and s and for categorical variables x_i is formulated using similar principles as in (Saves et al., 2023). Let $[R_i(\Theta_i)]$ be the GP correlation kernel for the i^{th} categorical variable c_i that depends on Θ_i, the hyperparameter matrix related to c_i. The variable c_i features L_i different levels and $[\Phi(\Theta_i)]$ is a chosen symmetric positive definite parameterization of the matrix Θ_i.

- If $\delta_i(x_r) = \delta_i(x_s) = 0$ (both variables are inactive): all the one-hot relaxed dimensions associated to this variable are also inactive meaning none of them is relevant.
- If $\delta_i(x_r) = \delta_i(x_s) = 1$ (both variables are active): all the one-hot relaxed dimensions associated to this variable are also active. Let c_i^r and c_i^s be the categorical variables associated to points r and s, taking respectively the ℓ_r^i and the ℓ_s^i level on the categorical variable c_i. If $\ell_r^i = \ell_s^i, [R_i(\Theta_i)]_{\ell_r^i,\ell_r^i} = 1$. Otherwise, assuming that the chosen kernel is the exponential kernel, this leads to the kernel

$$[R_i(\Theta_i)]_{\ell_r^i,\ell_s^i} = \exp\left(-\sqrt{2}[\Phi(\Theta_i)]_{\ell_r^i,\ell_r^i} - \sqrt{2}[\Phi(\Theta_i)]_{\ell_s^i,\ell_s^i}\right) \tag{12}$$

- If $\delta_i(x_r) \neq \delta_i(x_s)$ (only one of the variables is active): for all the relaxed dimensions associated to this variable, because of the algebraic distance, there is a induced distance 1 between both inputs. Assuming that the chosen kernel is the exponential kernel, this leads to the kernel

$$[R_i(\Theta_i)]_{\ell_r^i,\ell_s^i} = \exp\left(-\sum_{j=1}^{L_i}[\Phi(\Theta_i)]_{j,j}\right) \tag{13}$$

As shown in (Saves et al., 2022), continuous relaxation (one-hot encoding) generalizes the Gower distance. Therefore, denoting θ_{cov} the unique correlation parameter for the considered variable, we can define the following adaptation for the particular case in which we are using the Gower distance kernel for a categorical variable c_i with L_i

levels:

$$\begin{aligned}
&\bullet\ d_i(r,s) = 0 \text{ if both } r \text{ and } s \text{ are inactive.} \\
&\bullet\ d_i(r,s) = \sqrt{2}\ \theta_{\text{cov}} \text{ if both } r \text{ and } s \text{ are active.} \\
&\bullet\ d_i(r,s) = \frac{1}{2} L_i\ \theta_{\text{cov}} \text{ if either } r \text{ or } s \text{ are active.}
\end{aligned} \tag{14}$$

Finally, for the exponential homoscedastic hypersphere or for the homoscedastic hypersphere kernel we apply the imputation method: whenever one of the variables is inactive, it is treated as if its value would be the first available level L_0. These new categorical hierarchical kernels are implemented in SMT 2.3[1]. The rest of this work uses the Gower distance categorical hierarchical kernel for conditionally active variables defined in Equation (14).

We now discuss other stages of an optimization run where information about design space hierarchy can be leveraged.

5 Hierarchical Sampling and Correction Algorithms

In this section we investigate how to most effectively generate design vectors covering the design space in such a way as to obtain as much information about the design space behavior as possible for a given function evaluation budget. This is relevant both for sampling the design space and when correcting invalid design vectors. When sampling the design space (creating the DoE) for use as initial database of design points for SBO algorithms or initial population for evolutionary algorithms, the effects of rate diversity should be considered: if this effect is ignored, some regions in the design space may be over- or under-represented (Crawley et al., 2015). Correcting invalid design vectors might affect design space exploration by modifying results of evolutionary operators for evolutionary algorithms and infill optimization for SBO algorithms. Correction behavior is especially important for problems with high correction ratios due to the low change of randomly finding a valid design vector. First, we introduce the test problems used to investigate sampling and correction algorithms in Section 5.1. Then, sampling algorithms are investigated and selected in Section 5.2, and correction algorithms in Section 5.3.

5.1 Test Problems

To compare correction and sampling strategies, the follow test problems implemented in SBArchOpt are used (see Table 6 for detailed statistics):

- Three instances of a multi-stage rocket design problem ("Rocket"), adopted from García Sánchez (García Sánchez, 2024). The problem attempts to minimize cost while maximizing payload mass for a given target orbit altitude, by selecting the number of stages, engine types and amounts per stage, and geometrical parameters of stages and rocket head. The problem has been modified to be less tightly constrained than the original problem. In addition to this multi-objective problem ("Rocket"), we also use two single-objective versions: one that minimizes cost ("RCost") and one that minimizes a weighted function of cost and payload mass ("RWt"). The cost minimum lies in the group of single-stage rockets, which

Table 6: Test problems used for testing hierarchical sampling and optimization strategies.

Problem	SBArchOpt Class	n_f	n_g	n_{x_c}	n_{x_d}	$n_{valid,discr}$	IR	CR	CRF	MRD
RCost	SOLCRocketArch	1	2	6	8	18522	3.7	1.0	0%	94%
RWt	SOLCRocketArch	1	2	6	8	18522	3.7	1.0	0%	94%
Rocket	LCRocketArch	2	2	6	8	18522	3.7	1.0	0%	94%
SO GNC	SOMDGNCNoAct	1		6	11	327	150	16.1	56%	88%
FR GNC	SOMDGNCNoAct	1		6	11	327	150	16.1	56%	88%
Wt GNC	SOMDGNCNoAct	1		6	11	327	150	16.1	56%	88%
MD GNC	MDGNCNoAct	2		6	11	327	150	16.1	56%	88%
Jet SM	SimpleTurbofanArchModel	1	5	9	6	70	3.9	2.1	55%	60%

only makes up 0.3% of all valid discrete design vectors, whereas the weighted minimum lies in one of the much larger multi-stage rocket groups.

- Four instances of a Guidance, Navigation & Control (GNC) problem, adopted from (Crawley et al., 2015; Apaza and Selva, 2021), which features mass and failure rate as minimization objectives by selecting the number and type of, and connections between sensors, flight computers and actuators. The original problem has been modified to use continuous variables for selecting object types, to turn the problem into a mixed-discrete problem (compared to fully discrete). We use the mixed-discrete GNC problem version without actuators as a multi-objective test problem ("MD GNC"), a single-objective version that minimizes weight only ("Wt GNC"), a single-objective version that minimizes failure rate ("FR GNC"), and a single-objective version that solves a scalarized objective composed of weight and failure rate ("SO GNC"). The weight minimum lies in the group containing only one sensor and one computer, which is only represented by one x of $x_{valid,discr}$ (1 / 327 = 0.3%).
- A version of the simple jet engine problem solved in Section 7 that uses surrogate models for evaluation ("Jet SM"). This version represents the same optimization problem, however replaces the multidisciplinary thermodynamic cycle analysis evaluation by random forest regressors for each output, to reduce evaluation times (milliseconds, compared to minutes for the original problem). We use the random forest regressors implemented in scikit-learn[5]. Just as the original problem, this version features a hidden constraint (i.e. failed evaluations).

Table 6 provides some more statistics: both single-objective and multi-objective problems are included, and all except the GNC problems contain design constraints. The GNC problems feature a relatively high IR, showing that they indeed feature hierarchical design spaces. The GNC and Jet SM problems need correction algorithms as they get about half their IR from the need for correction shown by a CRF a little over 50%. MRD is a common occurrence in SAO problems too, as can be seen.

In subsequent sections, optimization performance is compared using the procedure described in Appendix A. A rank of 1 indicates best performance, higher values indicate progressively worse performance. The best performing infill is then selected by looking at the highest proportion of rank ≤ 2 and rank 1. Performance is compared

[5] https://scikit-learn.org/

based on ΔHV (Δ hypervolume) regret. ΔHV represents the distance to the known optimum (or Pareto front in case of multi-objective optimization) normalized to the range of objective values. Regret represents the cumulative ΔHV as seen over the optimization run. For more details, refer to Appendix A.

5.2 Sampling

Many sampling methods have been developed in the past, such as random sampling or Latin Hypercube Sampling (LHS) (Martins and Ning, 2022). Such methods can still be applied for hierarchical design spaces, for example by rejecting sampled design invalid design vectors (i.e. vectors with violated value constraints) as done in ConfigSpace. Instead of rejecting invalid design vectors, it is also possible to apply correction. Using conventional sampling methods with correction however might result in over- or under-representation of certain design space regions (Crawley et al., 2015). Another way is to sample design vectors directly from the set of valid design vectors $x_{valid,discr}$: this ensures generated design vectors are valid and therefore correction is not needed after sampling. To do this, $x_{valid,discr}$ may be divided into interesting subdivisions (e.g. based on subproblems as defined by dimensional variables (Pelamatti et al., 2020)) and sampling might be separated in two steps: first decide how many samples to draw from each subdivision, then sample from the subdivisions. As it is mostly the discrete design variables that decide the activeness of other design variables, in this work the hierarchical sampling procedure is applied to the discrete design vectors first. After sampling discrete design vectors, continuous design variables are filled by Sobol' sampling (Renardy et al., 2021), ignoring inactive continuous design variables.

One way of defining subdivisions is by using design variable activeness information. For example, groups can be defined based on the number of active design variables n_{act}, similar to what Kaltenecker et al. (Kaltenecker et al., 2019) did for sampling software product lines. An extension of this is to group by the active variables x_{act} directly. In that case, design vectors with the same number of active variables where these variables do not completely overlap are treated as different subdivisions. This approach is similar to how Frank et al. (Frank et al., 2016) define their architecture optimization problems: they divide by selections of values from a morphological matrix constrained by a compatibility matrix, then for each set of selections with the same active design variables they define a separate optimization problem. Table 7 shows how the valid discrete design vectors $x_{valid,discr}$ are separated into groups based on x_{act}.

Another way is to define subdivisions based on the value of certain discrete design variables. Pelamatti et al. (Pelamatti et al., 2020) defined subdivisions based on dedicated dimensional variables. This approach, however, requires the user to define which variables act as such grouping variables, thereby requiring the user to think in terms of the design variables rather than the architecture design space (i.e. domain-specific design space model). The goal of hierarchical sampling is to mitigate max rate diversity (MRD) issues: subdivisions can therefore be defined based on values of high-RD variables. Subdivisions are made by recursively selecting $\text{argmax} RD_j$, subject to $RD_j \geq RD_{min}$ where RD is determined for the current (partial) subdivision and RD_{min} is set such that only high-RD variables are used for grouping. Table 8 shows an example of grouping by MRD with $RD_{min} = 50\%$: group 3 is defined based on

Table 7: Grouping and weighting process of the hierarchical sampling algorithm: discrete design vectors are grouped by x_{act} and weighted by $w = n_{act}$ (number of active x); w_{rel} represents the relative weighting for sampling design vectors from groups (e.g. $w_{rel} = 36\%$ means 36% of x is sampled from that group). Red background indicates an inactive variable.

x_0	x_1	x_2	x_3	x_4	x_{act}	$w = n_{act}$	w_{rel}
0	0	0		0	x_0, x_1	4	36%
0	0	0		1	x_2, x_4		
0	0	0		2			
0	0	1	0		x_0, x_1	4	36%
0	0	1	1		x_2, x_3		
0	0	2	0				
0	0	2	1				
0	1				x_0, x_1	2	18%
1					x_0	1	9%

RD_0 (RD of x_0), group 1 and 2 are defined based on RD_1. No further groups are defined because the next-highest RD is RD_2, which is lower than RD_{min}. In this work we use $RD_{min} = 80\%$ as it is a good compromise between reducing group-internal MRD and the number of groups created.

After subdivisions are defined, it should be determined how many samples to take from each subdivision by assigning weights w to each group. This can be done using any distribution in principle, however we consider three to be most relevant for architecture optimization problems: uniform weighting ($w = 1$) as applied in (Kaltenecker et al., 2019), weighting by number of active design variables ($w = n_{act}$) as applied in (Pelamatti et al., 2020), or weighting by group size ($w = \sqrt{n_{x,grp}}$, where $n_{x,grp}$ is the number of x in the group). The reason for the latter two is that although smaller subdivisions (i.e. subdivisions with less active variables and therefore less possible discrete design vectors) should not be under-represented compared to larger subdivisions, larger subdivisions might need more samples to give a good overview of subdivision behavior to the optimization algorithm. Weighting by $n_{x,grp}$ uses $w = \sqrt{n_{x,grp}}$, because if $n_{x,grp}$ is used directly it is equivalent to hierarchical sampling without grouping. The square root is applied to prevent oversampling large groups. Table 7 shows how the valid discrete design vectors $x_{valid,discr}$ are separated into groups based on x_{act} and weights are assigned based on n_{act}. The relative weighting w_{rel} then determines how many of the requested samples are sampled from each group. For example, if 100 samples are requested, 36 will come from the first group, 36 from the second group, and 18 and 9 from the last two groups, respectively. If there are less discrete vectors in a group than requested, vectors can be selected multiple times if there are also continuous variables in the problem.

Table 8: Grouping by MRD: discrete design vectors are grouped by values of high-RD variables. The example uses $RD_{min} = 50\%$. Red background indicates an inactive variable.

	x_0	x_1	x_2	x_3	x_4
$x_j = 0$	89%	87%	42%	50%	33%
$x_j = 1$	11%	13%	29%	50%	33%
$x_j = 2$			29%		33%
RD	78%	74%	13%	0%	0%
Group 1	0	0	0		0
	0	0	0		1
	0	0	0		2
	0	0	1	0	
	0	0	1	1	
	0	0	2	0	
	0	0	2	1	
Group 2	0	1			
Group 3	1				

After valid discrete design vectors have been sampled, active continuous variables are assigned values using Sobol' sampling (Renardy et al., 2021). To summarize, the hierarchical sampling algorithm consists of the following steps (visualized in Table 7):

1. Generate all possible valid discrete design vectors $x_{valid,discr}$ and obtain associated activeness information δ;
2. Group design vectors (e.g. by active variables x_{act}) and weight each group (e.g. by the number of active variables n_{act});
3. Sample discrete design vectors according to the weights of each group;
4. Assign values to active continuous variables using Sobol' sampling.

Table 9 provides an overview of discussed sampling strategies for hierarchical design spaces. For problems with large design spaces, it might not be possible to generate $x_{valid,discr}$ due to memory or time limits. If this is the case, this excludes the use of the previously discussed hierarchical sampling algorithms, and therefore requires the use of a conventional sampling algorithm combined with a correction step.

First, the sampling strategies listed in Table 9 are tested on NSGA-II. We use the implementation of NSGA-II available in SBArchOpt as highlighted in Section 3.3. Problem-specific correction is used and $x_{valid,discr}$ is assumed to be available. NSGA-II is executed with a DoE size of $10 \cdot n_x$, 25 generations and 100 repetitions. Optimization performance is compared based on ΔHV regret using the procedure described in Appendix A. ΔHV (Δ hypervolume) represents the distance to the known optimum (or Pareto front in case of multi-objective optimization) normalized to the range of objective values. Table 10 presents the results of sampling strategies for NSGA-II. The hierarchical non-grouping sampling performs worst, especially on the rocket problems and weight-minimizing GNC problem (rank 8 and 455% penalty compared to the best strategy). This is because in those problems (part of) the optimum lies in architectures

Table 9: Overview of sampling strategies for hierarchical design spaces. $x_{valid,discr}$ refers to all valid discrete design vectors.

Strategy	Needs	Grouping	Weighting
Non-hierarchical	Correction		
Hierarchical	$x_{valid,discr}$	None	
		n_{act}	
		n_{act}	n_{act}
		n_{act}	$n_{x,grp}$
		x_{act}	
		x_{act}	n_{act}
		x_{act}	$n_{x,grp}$
		MRD	
		MRD	n_{act}
		MRD	$n_{x,grp}$

represented by only 0.3% of $x_{valid,discr}$ (1 sensor/computer for the GND problem; 1 stage for the Rocket problem). The non-grouping sampler uniformly samples over all $x_{valid,discr}$ and therefore has a high chance of not sampling these architectures. Problems with high MRD potentially suffer from this effect, depending on where the optimum lies. From the other samplers, the hierarchical samplers without weighting consistently perform better than samplers with weighting. Among the non-weighted hierarchical samplers, the n_{act} sampler performs best, with the other hierarchical samplers incurring between 6% and 7% relative penalty. Non-hierarchical sampling performs slightly worse than the best sampler, at an 12% mean performance penalty. Therefore, although the availability of $x_{valid,discr}$ improves algorithm performance, the non-availability does not prevent the problem from being solved.

For the BO algorithm, the non-hierarchical sampler and hierarchical non-weighted samplers are compared. The BO algorithm is executed with $n_{doe} = 3 \cdot n_x$, 40 infill points and 24 repetitions. For the Jet SM problem, $n_{doe} = 10 \cdot n_x$ for BO will be used, to correct for the fact that this problem features approximately a 60% failure rate and therefore needs a larger DoE. The weight-minimizing GNC problem ("Wt GNC") is not included, as it proved to easy for the BO algorithm to solve which lead to arbitrary differences in ΔHV regret. Table 11 presents sampling results for the BO algorithm. It shows the hierarchical x_{act} sampler performs best. The hierarchical n_{act} sampler performs worse with an 18% mean performance penalty. The mean performance penalty of the non-hierarchical sampler is 12%, just as for NSGA-II.

When comparing between NSGA-II and the BO algorithm, therefore, x_{act} and MRD remain as good candidates for hierarchical sampling. We select hierarchical sampling based on x_{act} for its slightly better performance on the BO algorithm. If $x_{valid,discr}$ is not available, the non-hierarchical sampler is used instead.

Table 10: Best performing sampling strategies on various test problems running NSGA-II, ranked by ΔHV regret (lower rank is better). Penalty represents the mean ΔHV regret increase compared to the best infill. Best performing strategy is underlined; darker colors represent better results. Abbreviation: grp. = grouping, wt. = weighting, hier. = hierarchical.

Strategy (grp.; wt.)	RCost	RWt	Rocket	SO GNC	FR GNC	Wt GNC	MD GNC	Jet SM	Rank 1	Rank ≤ 2	Penalty
Non-hier.	2	1	2	5	4	3	6	4	12%	38%	12%
Hier.	6	5	6	2	1	8	3	1	25%	38%	455%
Hier. (n_{act})	1	2	1	5	5	1	6	2	38%	62%	0%
Hier. (n_{act}; n_{act})	2	2	3	3	3	3	4	2	0%	38%	9%
Hier. (n_{act}; $n_{x,grp}$)	4	4	5	1	2	6	1	1	38%	50%	81%
Hier. (x_{act})	1	2	2	6	6	2	6	3	12%	50%	7%
Hier. (x_{act}; n_{act})	2	1	3	4	4	4	4	1	25%	38%	15%
Hier. (x_{act}; $n_{x,grp}$)	4	4	5	1	2	6	1	1	38%	50%	84%
Hier. (MRD)	1	2	1	4	4	3	5	1	38%	50%	6%
Hier. (MRD; n_{act})	3	3	4	3	3	5	3	1	12%	12%	25%
Hier. (MRD; $n_{x,grp}$)	5	4	5	1	2	7	2	1	25%	50%	118%

Table 11: Best performing sampling strategies on various test problems running the BO algorithm, ranked by ΔHV regret (lower rank is better). Penalty represents the mean ΔHV regret increase compared to the best infill. Best performing strategy is underlined; darker colors represent better results. Abbreviation: grp. = grouping, wt. = weighting, hier. = hierarchical.

Strategy (grp.; wt.)	RCost	RWt	Rocket	SO GNC	FR GNC	MD GNC	Jet SM	Rank 1	Rank ≤ 2	Penalty
Non-hier.	1	2	1	2	1	1	1	71%	100%	12%
Hier.	1	1	2	2	1	1	2	57%	100%	11%
Hier. (n_{act})	2	1	1	1	1	2	1	71%	100%	18%
Hier. (x_{act})	1	1	1	1	1	1	1	100%	100%	0%
Hier. (MRD)	1	1	2	1	1	2	1	71%	100%	2%

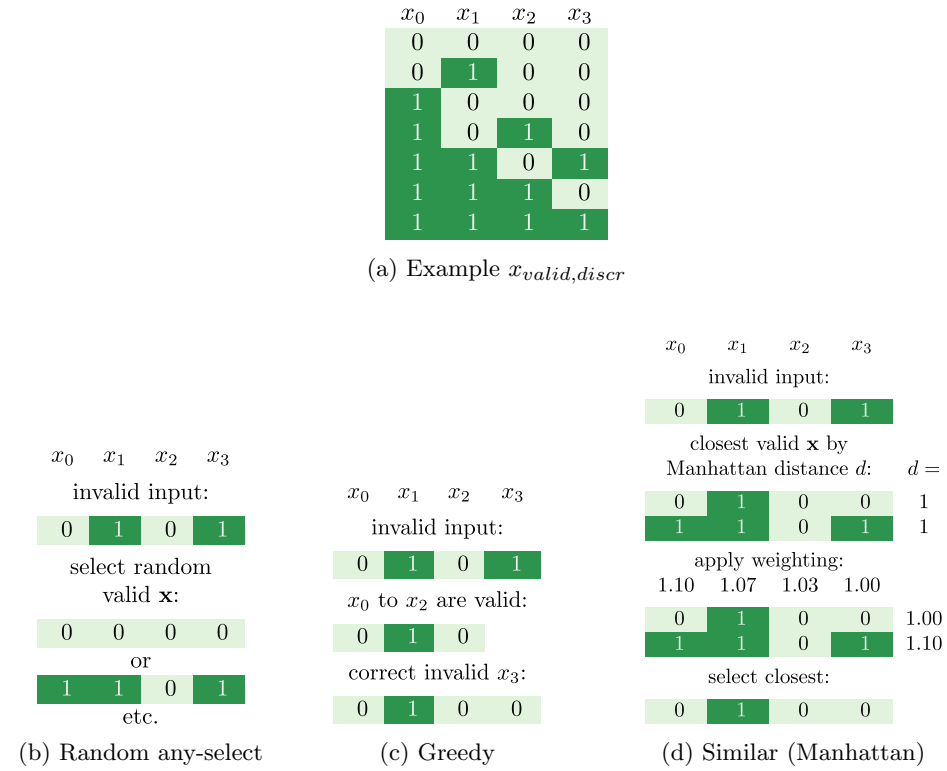
5.3 Correction

Correction is the operation of creating a valid design vector from an invalid design vector. Here we only discuss the process of correction for discrete design vectors, as we assume that only discrete variables determine the hierarchical structure in terms of activeness and value constraints. Continuous variables are therefore only subject to imputation if they are inactive.

The mechanism of correction depends on the hierarchical structure of the optimization problem and therefore can be implemented on a per-problem basis, as in (Bussemaker et al., 2021). There, design variables are corrected during parsing of the design vector into an architecture description: design variables not representing a valid option value are corrected one-by-one to the closest valid value. For example, when a design variable selects the third compressor stage for bleed off-take when only two compressor stages are available, the design variable is modified to select the second compressor stage for bleed off-take instead. This is called a greedy correction algorithm, as it takes the locally best choice for correcting invalid values for each invalid design variable. It is arguably the most convenient way to implement problem-specific correction, because of this step-by-step correction mechanism.

Problem-agnostic correction algorithms instead could reduce implementation effort and potentially improve optimizer performance. Such correction algorithms can be defined both for when $x_{valid,discr}$ is available and for when it is not: correction algorithms with $x_{valid,discr}$ available are called *eager* algorithms (see Figure 3), as they require the upfront exposure of valid design vectors; when $x_{valid,discr}$ is not available the correction algorithm is known as a *lazy* algorithm, as design vector validity is checked during the correction process rather than upfront. As eager correction algorithms have access to $x_{valid,discr}$, the simplest algorithm selects any of the available vectors as a replacement (called "Any-select"). We can define one that always returns the first (or any index for that matter) valid design vector, however that would not help with exploration at all: for problems with high imputation ratios there would still be a low chance to generate a new valid design vector. A better option therefore is to select a random valid design vector, as shown in Figure 3b.

Another approach is to select a design vector that is close to the design vector to be corrected x_{corr}. For eager algorithms this can be done in two ways: by applying a greedy algorithm or by selecting the most similar valid design vector. The eager greedy algorithm (see Figure 3c) repeatedly filters $x_{valid,discr}$ based on the selected value of each design variable, starting from the left of x_{corr}. If a given design variable value leaves no valid design vectors to be selected from, the closest value that does is selected. This is repeated until either only one valid design vector remains, or until all design variables have been processed. If in the latter case multiple design vectors remain, either the first (non-randomized configuration) or a random design vector is chosen (randomized configuration). Eager selection based on similarity (see Figure 3d) is done by calculating the weighted distances from x_{corr} to all vectors in $x_{valid,discr}$ and selecting the valid vector with the minimum distance to replace x_{corr}. Weighting factors linearly varying from 1.1 to 1.0 are applied in order to favor changes on the right side of the design vector over changes on the left, assuming that left-side design



(a) Example $x_{valid,discr}$

(b) Random any-select

(c) Greedy

(d) Similar (Manhattan)

Fig. 3: A visualization of different eager correction strategies

variables represent higher-impact decisions (Bussemaker and Ciampa, 2022). As distance metrics either Euclidean or Manhattan distance can be used. If multiple valid vectors have the same minimum distance to x_{corr} either the first (non-randomized configuration) or a random design vector (randomized configuration) can be selected.

Lazy correction algorithms do not have access to $x_{valid,discr}$ and instead provide some way to generate design vectors and check with a user-provided validation function $isCorrect(\boldsymbol{x})$ whether that design vector is valid or not. This generation and checking process continues until a valid design vector is found. Analogous to the any-selection eager algorithm, also an any-selection lazy algorithm can be defined. The difference in randomized and non-randomized selection however is that for non-randomized selection, the result may be remembered and returned in subsequent correction requests, whereas for randomized selection this is not the case. For randomized selection, an average of CR design vectors (see also the discussion in Section 2.1) will have to be generated before finding a valid design vector, which represents the main limitation of lazy over eager algorithms. Greedy selection is not possible for lazy algorithms, because $isCorrect(\boldsymbol{x})$ only operates on complete design vectors and not fractions of design vectors as would be needed for one-by-one correction. Lazy correction by similarity is done by modifying x_{corr} with some Δ_{corr} vector, which can either be generated

Table 12: Overview of correction algorithms for hierarchical design spaces. Euc and Manh refer to Euclidean and Manhattan distance metrics, respectively. See Figure 3 for a visualization of eager correction algorithms.

Type	Algorithm	Configuration	Needs
Problem-specific			Custom correction function
Eager	Any-select		$x_{valid,discr}$
	Greedy		$x_{valid,discr}$
	Similar	Distance (Euc/Manh)	$x_{valid,discr}$
Lazy	Any-select		$isCorrect(\boldsymbol{x})$
	Similar	Depth- or distance-first Distance (Euc/Manh)	$isCorrect(\boldsymbol{x})$

depth-first or distance-first. Depth-first Δ_{corr} generation directly applies the generated Cartesian product of Δ values for each design variable. Distance-first generation first generates all possible Δ_{corr} vectors and then sorts them by Euclidean or Manhattan distance before applying them to x_{corr}. The same distance-weighting process as for eager similarity selection is used here. ConfigSpace uses a randomized version of lazy similarity correction: candidate design vectors are generated by replacing one design variable by a random valid value at a time. We do not consider this method, as it does not allow to correct multiple design variables at a time. Table 12 presents an overview of discussed correction strategies.

The identified correction strategies are tested on NSGA-II first, with a DoE size of $10 \cdot n_x$, 25 generations and 100 repetitions. Eager correction strategies are tested with the hierarchical x_{act} sampling algorithm; lazy strategies with the non-hierarchical sampler as here the assumption is that $x_{valid,discr}$ is not available. Correction strategies are tested for the various algorithm-specific configuration options (see Table 12). Eager and lazy correction are additionally compared to problem-specific correction. Table 13 presents results of correction strategies for NSGA-II. Eager correction performs better than lazy correction, and similarly to problem-specific correction. Eager Any-select performs best, with Eager similar (Manhattan or Euclidean distance) performing similarly. Lazy similar (depth-first) correction performs best among lazy correction strategies. Lazy correction, however, takes between 1 and 2 orders of magnitude longer than problem-specific and eager correction: 2 to 50 ms, compared to 0.1 to 1 ms for eager and problem-specific correction. Additionally, lazy correction time increases linearly with CR, because it is based on a trial-and-error approach.

Table 13: Best performing correction strategies on various test problems running NGSA-II, ranked by ΔHV regret (lower rank is better). Penalty represents the mean ΔHV regret increase compared to the best infill. Best performing strategy is underlined; darker colors represent better results.

Correction	Config	Sampling	RCost	RWt	Rocket	SO GNC	FR GNC	Wt GNC	MD GNC	Jet SM	Rank 1	Rank ≤ 2	Penalty
Eager Any-select		Hier. x_{act}	1	1	1	1	1	4	2	1	75%	88%	0%
Eager Greedy		Hier. x_{act}	1	2	1	3	5	1	4	1	50%	62%	5%
Eager Similar	Manh	Hier. x_{act}	1	1	1	1	3	2	1	2	62%	88%	3%
Eager Similar	Euc	Hier. x_{act}	1	1	2	2	4	2	1	2	38%	88%	3%
Lazy Any-select		Non-hier.	1	1	2	4	3	1	3	3	38%	50%	8%
Lazy Similar	Depth-f.	Non-hier.	2	1	2	3	3	2	2	2	12%	75%	8%
Lazy Similar	Manh; Dist-f.	Non-hier.	3	1	3	3	4	3	1	3	25%	25%	15%
Lazy Similar	Euc; Dist-f.	Non-hier.	3	2	3	3	3	1	1	3	25%	38%	12%
Problem-specific		Hier. x_{act}	2	2	2	3	4	1	3	2	12%	62%	6%
Problem-specific		Non-hier.	3	1	2	2	2	3	2	3	12%	62%	8%

Table 14: Best performing correction strategies on various test problems running the BO algorithm, ranked by ΔHV regret (lower rank is better). Penalty represents the mean ΔHV regret increase compared to the best infill. Best performing strategy is underlined; darker colors represent better results.

Correction	Config	Sampling	RCost	RWt	Rocket	SO GNC	FR GNC	MD GNC	Jet SM	Rank 1	Rank ≤ 2	Penalty
Eager Any-select		Hier. x_{act}	1	1	1	2	1	2	1	71%	100%	2%
Eager Similar	Manh	Hier. x_{act}	1	1	2	2	1	1	1	71%	100%	3%
Lazy Similar	Depth-f.	Non-hier.	1	1	1	2	1	2	2	57%	100%	7%
Problem-specific		Hier. x_{act}	1	2	1	1	1	1	1	86%	100%	0%
Problem-specific		Non-hier.	1	2	1	2	2	1	1	57%	100%	12%

For the BO algorithm, best performing eager and lazy correction algorithms and problem-specific correction are compared. The BO algorithm is executed with $n_{doe} = 3 \cdot n_x$ ($n_{doe} = 10 \cdot n_x$ for the Jet SM problem), 40 infill points and 24 repetitions. Table 14 presents sampling results for the BO algorithm. Problem-specific correction with hierarchical sampling performs best. Eager correction performs similar to problem-specific correction, however attains rank 1 less often. Lazy correction and problem-specific correction with non-hierarchical sampling perform worst. Based on these investigations, we conclude that problem-specific correction is sufficient for good optimizer performance, both for NSGA-II and BO. If the user prefers not to implement problem-specific correction instead, and if $x_{valid,discr}$ is available, then Eager Any-select correction should be used. These results are used in the next section where the influence of hierarchy information integration strategies are investigated.

6 Hierarchical Design Space Integration Strategies

According to Zaefferer et al. (Zaefferer and Horn, 2018) there are three high-level strategies for considering design space hierarchy when implementing and solving an optimization problem:

1. *Naive*: no modification of the optimization algorithm at all, thereby effectively ignoring the effect of design variable hierarchy;
2. *Correction and imputation*: here correction and imputation are applied to ensure that all evaluated design vectors are valid (see also Section 2.1);
3. *Explicit consideration*: the hierarchical structure is explicitly made available to and used by the optimization algorithm.

The naive approach means that there might be a discrepancy between which design vectors the optimizer thinks are being evaluated and which design vectors actually are evaluated. For example, design vectors where only inactive design variables differ in value all represent the same system architecture instance. Not making this information available to the optimizer might lead to wasted computational resources, because exploration could be performed in sections of the design space that have no influence on performance. The optimization might also stall if the imputation ratio (see Section 2.1) is too high, because of the low chance of randomly finding valid design vectors. As discussed in Section 2.1, *correction and imputation* ensure that design vectors are valid by ensuring value constraints are satisfied and setting inactive design variables to some default value, respectively. Applying these operations avoids the aforementioned design vector mapping problems and only requires that the optimization algorithm accepts modified design vectors as output of the evaluation call. Another way to support correction and imputation is through an *ask-and-tell* interface (Collette et al., 2013): here a process external to the optimizer has control over the optimization loop, "asking" the optimizer for one or more design vectors to evaluate and "telling" the optimizer the results after evaluation is finished. Results are "told" to the optimizer together with the corresponding design vectors, which means the ask-and-tell pattern allows integrating correction and imputation steps without any further modifications. Therefore, any optimization framework or algorithm that implements an ask-and-tell interface is compatible with correction and imputation, for example pymoo (Blank

and Deb, 2020), BoTorch (Balandat et al., 2019), Trieste (Picheny et al., 2023) and HEBO (Cowen-Rivers et al., 2020).

Correction and imputation can also be implemented using a *repair* operator (Salcedo-Sanz, 2009): a problem-specific function that modifies design vectors, for example to satisfy an a-priori constraint or otherwise improve the design points using heuristics. The advantage of a repair operator over ask-and-tell or modifying design vectors in the evaluation function is that the correction and imputation operators are now available as a standalone function rather than always tied to evaluation. This allows correction and imputation to be applied during other steps in the optimization process than only evaluation, for example when generating initial design points or when searching the design space for the best infill point for surrogate-based optimization algorithms (Garrido-Merchán and Hernández-Lobato, 2020).

The most invasive way of supporting hierarchical design spaces is through *explicit consideration* by optimization algorithms. Here correction and imputation become an integral part of the optimizer and *activeness* information as defined by the δ-function (see Section 2.1) is made available to the optimizer. The availability of activeness information makes the following possible:

- Generating all possible valid design vectors $x_{valid,discr}$, and therefore using hierarchical sampling and problem-agnostic correction algorithms, as investigated in Section 5.
- Using hierarchical kernels in GP models used by BO (Hutter and Osborne, 2013; Lu et al., 2018; Levesque et al., 2017; Saves et al., 2024).

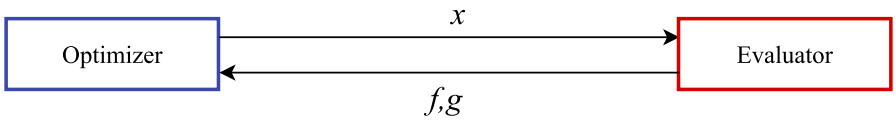
Figure 4 compares the three high-level integration strategies. It highlights the existence of a standalone corrector function that corrects and imputes design vectors and optionally returns activeness information.

One way to explicitly consider the hierarchical structure is by formally modeling the hierarchical structure and making this model available to the optimization algorithm. It for example enables problem decomposition approaches (Frank et al., 2016; Pelamatti et al., 2020) and the development of dedicated evolutionary search operators (Abdelkhalik, 2012; Nyew et al., 2015; Apaza and Selva, 2021). Additionally, because the model provides all information needed without needing to interrogate the problem definition (i.e. as needed for a repair operator) this opens up the possibility for physically separating the optimizer from the function evaluation, for example to enable remote ask-and-tell execution. Hierarchical design space models can be classified according to different levels of complexity:

1. Single-level: one set of variables determining activeness of a disjoint set of conditional variables, e.g. (Pelamatti et al., 2020; Audet et al., 2023);
2. Tree-structured: conditional variables can also determine activeness of other variables, e.g. BoTorch (Balandat et al., 2019);
3. Directed acyclic graph: activeness can be determined by multiple variables, e.g. (Hutter and Osborne, 2013) and ConfigSpace (Lindauer et al., 2019);
4. Directed graph: also supports cyclic dependencies, e.g. the Architecture Design Space Graph (ADSG) (Bussemaker and Ciampa, 2022).

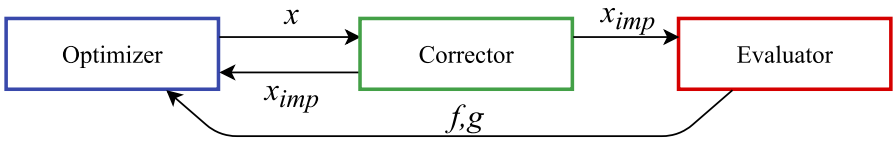
1: Naive

The hierarchical structure of the optimization problem is ignored.



2: Correction & Imputation

x is corrected and imputed before evaluation and x_{imp} is returned to the optimizer.
The corrector may be made available as a standalone repair operator.



3: Explicit Consideration

The corrector is available as a standalone repair operator
and returns activeness information δ in addition to x_{imp}.

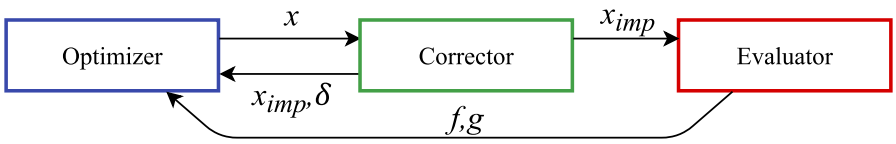


Fig. 4: High-level strategies for dealing with hierarchical optimization

ConfigSpace[6] implements one of the most general hierarchical design space definitions known to the authors. ConfigSpace is an open-source Python library that supports the definition of mixed-discrete design variables, conditional activation of design variables, and value constraints (Lindauer et al., 2019). Conditional activation is specified by if-clauses that can be composed of equality and inequality checks and Boolean conjunctions (AND, OR). Conditional activation depending on multiple variables is supported, as long as there are no cyclic dependencies. Value constraints can be specified using forbidden clauses: generic if-clauses between two or more design variables that define a violated value constraint when the condition is true. Once defined, the design space object can then be used to query which design variables are conditionally active, query whether a given design vector is valid, correct and impute a design vector, get activeness information for a given design vector, and generate random valid design vectors. ConfigSpace is used by several optimization frameworks

[6] https://automl.github.io/ConfigSpace/

Table 15: Different strategies for dealing with design variable hierarchy in optimization algorithms.

General strategy	Integration	Usage	1	2	3	4	5	6
Naive	N/A	N/A						
Correction & Imputation	x-output	Evaluation	✓					
	Ask-and-tell	Evaluation	✓					
	Repair operator	Evaluation, sampling	✓	✓				
Explicit consideration	Activeness $\delta_i(\mathbf{x})$	Sampling	✓	✓	✓			
	Activeness $\delta_i(\mathbf{x})$	Sampling & modeling	✓	✓	✓	✓		
	Formal model	Sampling & modeling	✓	✓	✓	✓	✓	✓

[1]All evaluated x are valid.

[2]All x in sampling and infill search are valid.

[3]Availability of all valid discrete design vectors $x_{valid,discr}$.

[4]Hierarchical kernels for surrogate modeling.

[5]Dedicated search operators and possibility for problem decomposition.

[6]Physical separation between optimization and evaluation code.

to support hierarchical design spaces, for example SMAC3 (Lindauer et al., 2022), BOAH (Lindauer et al., 2019), OpenBox (Li et al., 2021) and SMT (Saves et al., 2024). It should be noted that also if the optimization problem does not expose an explicit hierarchical design space model, the problem still contains some model of the hierarchical structure, either explicitly defined or implicitly embedded in the evaluation code (e.g. as for the turbofan optimization benchmark problem of (Bussemaker et al., 2021)). It might be more convenient for the user to think in terms of a domain-specific model of the design space compared to directly thinking in terms of design variables, and thus it might help adoption of architecture optimization. Examples of design space modeling techniques developed in the context of systems engineering and architecture optimization include the Architecture Decision Graph (Simmons, 2008), the Adaptive Reconfigurable Matrix of Alternatives (Mavris et al., 2008), the Architecture Decision Diagram (Apaza and Selva, 2021), feature models (Benavides et al., 2010; Czarnecki et al., 2012), function-means models (Gedell and Johannesson, 2012), and the Architecture Design Space Graph (ADSG) (Bussemaker and Ciampa, 2022). Retiarii (Zhang et al., 2020) and NASBench (Ying et al., 2019) provide domain-specific formats for modeling deep neural network architecture design spaces (Elsken et al., 2019). Table 15 presents a detailed overview of discussed integration strategies, and capabilities gained at each level of integration.

The correction and sampling algorithms developed in Section 5 need the highest level of hierarchy integration: activeness information. We now compare the different levels of integration to investigate if applying higher levels of integration (see Table 15) indeed results in better optimizer performance. Ask-and-tell integration has the same results as x-output integration: they only differ in the way the algorithm is called. Similarly, having a formal model changes nothing in the information available to the algorithm compared to the activeness integration. Therefore, experiments are only run for the naive, x-output, repair and activeness integration strategies. For

Table 16: Tested hierarchy integration strategies and impact on available capabilities. General strategy refers to strategies presented in Table 15. Abbreviations: hier. = hierarchical, corr. = correction, expl. cons. = explicit consideration.

	Naive	X out	Repair	Hier. sampling	Activeness
General strategy	Naive	Corr.	Corr.	Expl. cons.	Expl. cons.
Valid x-output		✓	✓	✓	✓
Repair operator			✓	✓	✓
Hierarchical sampling				✓	✓
Hierarchical GP (BO only)					✓

BO, the activeness strategy is run in two configurations: one where activeness only used for hierarchical sampling, and one where activeness is available additionally for the GP models. An overview of tested integration strategies and available capabilities for each strategy is provided in Table 16. NSGA-II is executed with $n_{doe} = 10 \cdot n_x$, 25 generations and 100 repetitions. The BO algorithm is executed with $n_{doe} = 3 \cdot n_x$ ($n_{doe} = 10 \cdot n_x$ for the Jet SM problem), 40 infill points and 16 repetitions. The hierarchical test problems presented in Section 5.1 and shown in Table 6 are used to compare the hierarchical optimization strategies.

Table 17 shows that for NSGA-II, the level of integration does not influence performance much. Repair performs best, indicating that hierarchical sampling (as used in Activeness) is not necessarily beneficial. Table 18 shows that for BO a higher level of integration improves optimizer performance. Naive, X-out and Repair integration is penalized significantly (134%, 149% and 89%, respectively), showing that in these cases the BO algorithm is less well able to suggest valid infill design points. If we consider only querying a Gaussian Process (GP) model, as shown by Saves et al. in (Saves et al., 2024), taking into account the activeness in the GP using hierarchical kernels leads to better results than using non-hierarchical kernels. However, Table 18 shows a reduction in performance when using hierarchical kernels (Activeness) compared to non-hierarchical kernels (Hierarchical sampling). This result is due to two factors: the potentially high concentration of training points in some areas of the design space, and the structure of the used test problems. High concentrations of training points in certain areas of the design space occur because the algorithm is attempting to close-in on an optimum in these areas, and is a common occurrence in SBO. The Rocket and GNC problems have relatively smooth objective functions, and therefore in their hierarchical versions the non-hierarchical GP models are still able to accurately represent the objective functions. It is expected that the first effect will be reduced for higher-dimensional problems, and that the second effect will be reduced for problems with more non-linear and non-smooth objective and constraint functions. Therefore we keep both the non-hierarchical and hierarchical GP models into consideration for the subsequent more complex investigation.

Table 17: Comparison of hierarchical optimization strategies on various test problems running NSGA-II, ranked by ΔHV regret (lower rank is better). Penalty represents the mean ΔHV regret increase compared to the best infill. Best performing strategy is underlined; darker colors represent better results.

	RCost	RWt	Rocket	SO GNC	FR GNC	Wt GNC	MD GNC	Jet SM	Rank 1	Rank ≤ 2	Penalty
Naive	2	2	3	1	3	2	1	1	38%	75%	-3%
X out	3	1	2	2	1	2	3	2	25%	75%	1%
Repair	1	1	2	2	2	2	2	2	25%	100%	0%
Activeness	1	1	1	3	4	1	4	1	62%	62%	3%

Table 18: Comparison of hierarchical optimization strategies on various test problems running the BO algorithm, ranked by ΔHV regret (lower rank is better). Penalty represents the mean ΔHV regret increase compared to the best infill. Best performing strategy is underlined; darker colors represent better results.

	RCost	RWt	Rocket	SO GNC	FR GNC	MD GNC	Jet SM	Rank 1	Rank ≤ 2	Penalty
Naive	2	2	2	1	1	3	2	29%	86%	134%
X out	2	2	2	3	2	4	2	0%	71%	149%
Repair	2	2	1	2	1	2	2	29%	100%	89%
Hier sampl.	1	1	1	1	1	1	1	100%	100%	0%
Activeness	1	1	1	1	1	3	1	86%	86%	20%

7 Application: Jet Engine Architecture

To demonstrate the application of the architecture optimization strategies presented in this work, a jet engine optimization problem is solved using a Bayesian Optimization (BO) algorithm. The jet engine optimization problem is a benchmark problem specifically developed to provide realistic architecture optimization challenges and behavior (Bussemaker et al., 2021). It is defined using the jet engine optimization testing framework presented by Bussemaker et al. (Bussemaker et al., 2021), the purpose of which is the provide a flexible way to define jet engine optimization problems with the purpose of testing optimization algorithms for SAO, see Figure 5 for an overview. The user defines the optimization problem by selecting from available architectural choices (that define the design variables) and metrics and inputting the flight conditions and power offtakes to size the engine for. Available architectural choices include whether to add a fan and bypass flow, the number of compressor and turbine stages, the use of intercooling and inter-turbine burning, and where to apply bleed and power offtakes. The selected choices are used to define the continuous, integer, and categorical design variables for the optimizer. A translator code is provided that translates a design vector generated by the optimizer into an architecture instance defined using objects. These objects contain all information required to build the analysis problem, including input parameters (from flight conditions, offtake requirements, or design vectors) and airflow connection sequences (e.g. compressor to combustor, combustor to turbine, etc.). A builder code then takes these objects and constructs an OpenMDAO (Gray et al., 2019) problem that performs thermodynamic cycle analysis and engine sizing using pyCycle (Hendricks and Gray, 2019), with the main output being the Thrust-Specific Fuel Consumption (TSFC) of the engine. Handbook methods are added to calculate additional metrics such as noise level, NOx emissions, weight, and size. Thermodynamic cycle analysis takes between 1 and 5 minutes to complete. However, it is not guaranteed to converge to a feasible solution, leading to the presence of a hidden constraint. If the hidden constraint is violated, metrics are set to NaN (not a number). The testing framework enables specification of a wide variety of test problems, all based on realistic engineering behavior, however with varying number of design variables, objectives, and constraints. The code is available open source[7], for more information the reader is referred to (Bussemaker et al., 2021).

In this investigation, we use the simple problem formulation defined in (Bussemaker et al., 2021):

[7] https://github.com/jbussemaker/OpenTurbofanArchitecting/

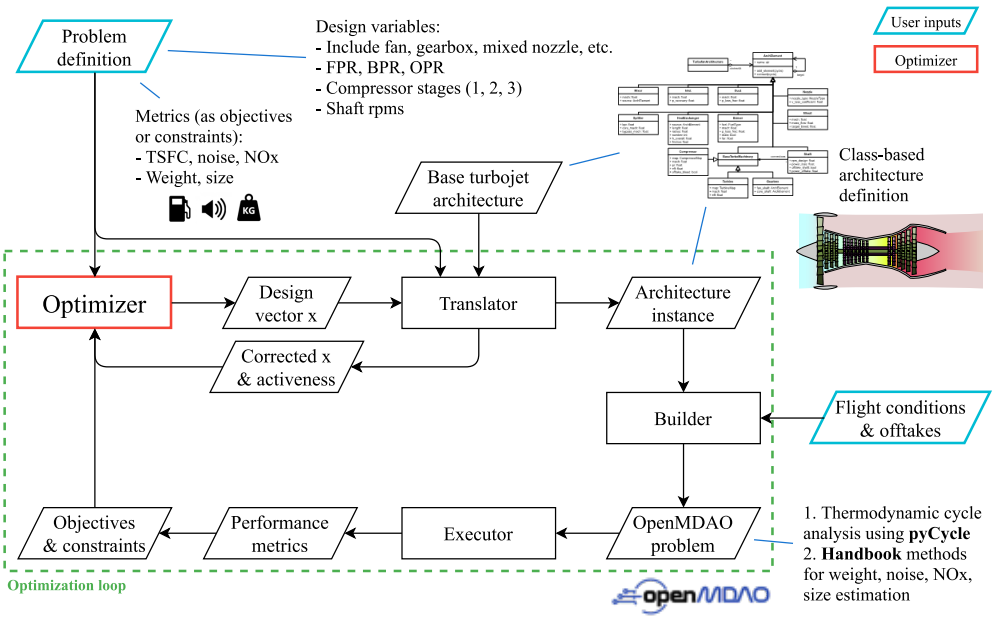


Fig. 5: Overview of the jet engine optimization testing framework. The user provides the problem definition (in terms of design variables and metrics selected from a database) and the flight conditions and power offtakes to size the engine for. Engine schematic adapted from original by K. Aainsqatsi, available at https://commons.wikimedia.org/wiki/File:Turbofan_operation.svg.

$$
\begin{aligned}
&\textit{minimize} && TSFC, && \\
&\textit{w.r.t.} && IncludeFan \in \{False, True\} && \\
& && 2.0 \leq BPR \leq 12.5 && \\
& && 1.1 \leq FPR \leq 1.8 && \\
& && n_{shafts} \in \{1, 2, 3\} && \\
& && 1.1 \leq OPR \leq 60.0 && \\
& && 0.1 \leq PR_{factor,i} \leq 0.9 && i = 2, 3 \\
& && 1000 \leq RPM_i \leq 20000 && i = 1, 2, 3 \\
& && IncludeGearbox \in \{False, True\} && \\
& && 1.0 \leq GearRatio \leq 5.0 && \\
& && MixedNozzle \in \{False, True\} && \\
& && PowerOfftake \in \{1, 2, 3\} && \\
& && BleedOfftake \in \{1, 2, 3\} && \\
&\textit{subject to} && M_{jet} \leq 1.0 && \\
& && PR_{factor,sum} \leq 0.9 && \\
& && PR_{max,i} \leq 15.0 && i = 1, 2, 3
\end{aligned}
\tag{15}
$$

which is a single-objective (TSFC minimization) problem with several architectural choices: fan inclusion $IncludeFan$, number of compressor stages n_{shafts}, gearbox inclusion $IncludeGearbox$, mixed nozzle selection $MixedNozzle$ and power offtake

Table 19: Comparison of median optimal TSFC (minimization) values achieved for the jet engine problem. Refer to Table 16 for comparison of hierarchy integration strategies. ΔHV regret was not available for NSGA-II with 3250 evaluations.

Algorithm	Hierarchy integration	N_{fe}	TSFC [g/kNs]	ΔHV regret
BO	Activeness	300	6.633	0.91
BO	Hierarchical sampling	300	6.653 (+0.30%)	0.86 (-4.8%)
BO	Repair	300	6.640 (+0.11%)	1.59 (+75%)
NSGA-II		300	7.455 (+12.4%)	3.67 (+305%)
NSGA-II		3250	6.640 (+0.11%)	–

locations $PowerOfftake$ and $BleedOfftake$. The problem includes several levels of activation hierarchy: bypass ratio BPR, fan pressure ratio FPR, gearbox inclusion $IncludeGearbox$ and mixed nozzle selection $MixedNozzle$ are only active if the fan is included ($IncludeFan = True$); the gear ratio $GearRatio$ is only active if $IncludeGearbox = True$; and shaft-related pressure ratio factors $PR_{factor,i}$ and rotational speeds RPM_i are only active if the respective number of shafts are selected. The power offtake $PowerOfftake$ and bleed offtake $BleedOfftake$ selections are always active, however are value-constrained by the available shafts.

In total, there are 70 valid discrete design vectors. However, the Cartesian product of discrete variables leads to 216 combinations: the discrete imputation ratio therefore is $IR_d = 216/70 = 3.1$ (see Eq. (1)). The continuous imputatio ratio $IR_c = 1.26$ (see Eq. (2)), meaning that there are on average $9/1.26 = 7.14$ continuous variables active (as seen over all valid discrete design vectors). The overall imputation ratio is $IR = 3.89$ (see Eq. (3)). The correction ratio $CR = 2.10$ (see Eq. (6)), which leads to correction ratio fraction $CRF = 55\%$ (see Eq. (7)). Thus, a little over half of the design space hierarchy is due to value constraints (i.e. the need for correction). The problem additionally features 5 design constraints, constraining the output jet Mach number M_{ket} and pressure ratio distributions over the selected compressor stages ($PR_{factor,sum}$ and $PR_{max,i}$). Additionally, the underlying thermodynamic cycle analysis and sizing code does not always converge, leading to a hidden constraint being violated in approximately 50% of design points generated in a random DoE. We use the problem implemented in SBArchOpt as SimpleTurbofanArch.

The BO algorithm is executed 24 times with $n_{doe} = 113$, $n_{infill} = 187$ (a budget of 300 evaluations), and $n_{batch} = 4$. The algorithm is executed for (see Table 16) Repair integration (no hierarchy information exposed, however the repair operator is available), Hierarchical Sampling (therefore an MD GP used during the optimization), and Activeness (both hierarchical sampling and hierarchical GP is used). The effectiveness of NSGA-II has already been demonstrated in (Bussemaker et al., 2021), however for completeness we compare BO results against NSGA-II results. NSGA-II is executed with repair operator available and using hierarchical sampling.

Figures 6 and 7 present the results of the jet engine optimization for the BO algorithm with the three compared hierarchy integration strategies. Table 19 presents achieved median optimal TSFC and ΔHV regret values. All BO algorithm configurations approximate the previously-found optimum within 0.2%, or slightly improve it,

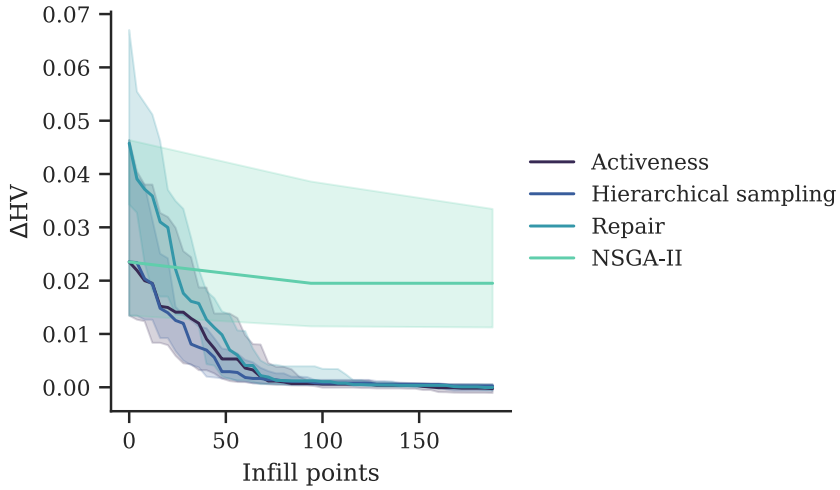


Fig. 6: Comparison of NSGA-II to the BO algorithm with different levels of hierarchy integration (see also Table 16): Repair (no hierarchy information; repair operator available), Hierarchical sampling (no hierarchical GP) and Activeness (hierarchical sampling and hierarchical GP)

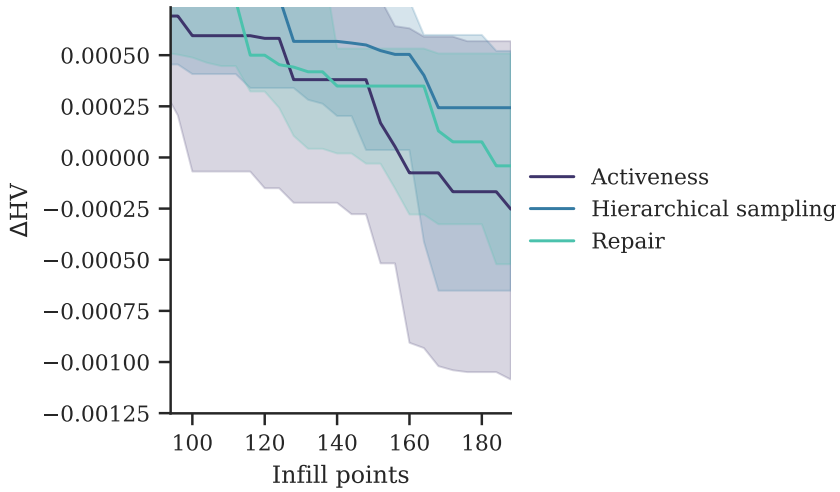


Fig. 7: Details of final optimization phase of comparison shown in Figure 6, only showing the BO algorithm

within the 300 function evaluations. The previously-found optimum was found with NSGA-II and an evaluation budget of 3250; BO therefore can be considered to be able to find the same result in 92% less function evaluations. For the BO algorithm, Activeness and Hierarchical sampling perform with similar ΔHV regret (see Table 19). However, Activeness is able to find a slightly better TSFC. Repair yields a significantly higher ΔHV regret due to less effective initial sampling, however in the end finds a better TSFC than Hierarchical sampling. NSGA-II is not able to improve much upon its initial population within 300 function evaluations. These results demonstrate that BO can be used to solve SAO problems, enabling a significant reduction in number of function evaluations needed compared to evolutionary algorithms like NSGA-II. Integration of design space hierarchy information, both by using the hierarchical sampling algorithm and using hierarchical GP, results in better optimizer performance.

8 Conclusions and Outlook

System Architecture Optimization (SAO) problems are challenging to solve due to their mixed-discrete and hierarchical design spaces combined with constrained, multi-objective, black-box, and expensive to evaluate solution spaces that potentially are subject to hidden constraints. Four metrics are defined for characterizing SAO design space hierarchy: imputation ratio IR, correction ratio CR, correction fraction CRF, and maximum rate diversity MRD. The mixed-discrete, black-box nature of SAO problems require the use of gradient-free, global optimization algorithms. Multi-objective Evolutionary Algorithms (MOEA), in particular NSGA-II, have been shown in the past to be effective, although at a high function evaluation cost. The potential of Surrogate-Based Optimization (SBO) and in particular *Bayesian Optimization (BO)* algorithms is identified, mainly due to their high potential in solving global optimization problems with a restricted function evaluation budget.

This work extends a non-hierarchical BO algorithm to support optimization in hierarchical SAO design spaces. First, hierarchical Gaussian Process (GP) kernels developed in previous research and used in the SMT library are extended to also support *categorical hierarchical variables*. This enables the usage of hierarchical GP models in any mixed-discrete hierarchical design space.

Then, hierarchical sampling and correction algorithms are developed to deal with rate diversity and correction ratio effects, respectively. The best performing and thereafter applied *hierarchical sampling* algorithm groups discrete design vectors by active design variables x_{act}, then uniformly samples discrete design vectors from the defined groups, and applies Sobol' sampling to generate samples for continuous variables. Hierarchical sampling performs better than non-hierarchical sampling, however cannot be used if valid discrete design vectors $x_{valid,discr}$ are not available. *Problem-agnostic correction algorithms* are developed to deal with imputation ratio effects. Eager correction algorithms have access to valid discrete design vectors $x_{valid,discr}$, whereas lazy correction algorithms do not. Using problem-agnostic correction does not necessarily lead to better optimizer performance when compared for both NSGA-II and BO, and therefore it is not applied in the rest of the work.

Information about the hierarchical design space can be integrated into an optimization algorithm at various levels: the information can be ignored (naive approach), design vectors can be modified either by the evaluation function or by a standalone repair operator (correction and imputation approach), and activeness information can additionally be made available and used by the optimizer (explicit consideration approach). It is shown that for NSGA-II the level of integration does not influence optimizer performance much. For BO, higher levels of integration increase optimizer performance, except that using a hierarchical GP for infill search reduces performance.

The developed BO algorithm is demonstrated on a jet engine optimization problem that features design space hierarchy, design constraints, and hidden constraints. It is shown that BO is able to find the optimum in 92% less function evaluations than NSGA-II. The higher levels of hierarchy integration perform better. The developed BO algorithm is implemented as as ArchSBO in SBArchOpt[8] (Bussemaker, 2023). Used test problems are also available in SBArchOpt.

BO has been shown to be effective at solving moderate-size architecture optimization problems. Use of BO should also be enabled for larger design space sizes, i.e. tens to hundreds of design variables (Donelli et al., 2023), for example using dimension reduction techniques such as EGORSE (Priem et al., 2023). Multi-fidelity BO may also be considered to make BO more effective and/or further reduce required computational resources. To aid adoption of SAO, it should be easy to formulate and run optimizations. Formulating architecture optimization problems can be made easier by developing and applying modeling techniques that do not require the explicit definition of design variables and design variable hierarchy, e.g. (Bussemaker and Ciampa, 2022; Zhang et al., 2020). Selecting and configuring optimization algorithms has been made easier by implementing all results of this research in the SBArchOpt library and making it open-source. Implemented algorithms are provided with default configurations that should be appropriate for a wide range of optimization problems. Running optimizations can be made easier by enabling configuration and execution of optimization algorithms from the architecture modeling environment directly, rather than requiring the user to switch environments and/or write additional code.

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. The authors would like to recognize Luca Boggero and Björn Nagel for supporting the secondment of Jasper Bussemaker at ONERA DTIS in Toulouse, during which the research presented in this paper was mostly performed.

[8] https://sbarchopt.readthedocs.io/

Replication of Results

All test problems used in this work are available in the open-source SBArchOpt library. Code used to run the experiments is available open-source too, at github.com/jbussemaker/ArchitectureOptimizationExperiments (hidden-constraints branch). The repository maps generated results (Tables and Figures) to executable Python functions to enable reproducing results. Developed results have been implemented in SBArchOpt version 1.4.

Appendix A Optimizer Performance Comparison Method

This appendix describes the method for comparing optimization algorithm performance by ranking various algorithm configurations based on ΔHV regret. ΔHV (Δ hypervolume) represents the distance to the known optimum (or Pareto front in case of multi-objective optimization) normalized to the range of objective values. For single-objective optimization, the range is calculated from the difference between the known optimum and the maximum objective value encountered during the optimization; for multi-objective optimization the hypervolume of the known Pareto front is taken as normalization value. To compare multiple optimization runs with different initial Design of Experiments (DoEs), $\Delta HV_{ratio} = \Delta HV / \Delta HV_0$ is used, where ΔHV_0 is the value at the first iteration (i.e. after the DoE has been evaluated). Regret represents the cumulative error, in our case ΔHV_{ratio}, over the course of an optimization (Garnett, 2023): lower values are better, and the value gives an indication of both how closely the optimum is approach by the end of the optimization and by how quickly this was achieved. ΔHV regret at iteration i is calculated from:

$$Regret(\Delta HV)_i = Regret(\Delta HV)_{i-1} + \frac{1}{2} n_{step} \left(\Delta HV_{ratio,i-1} + \Delta HV_{ratio,i}\right) \quad \text{(A1)}$$

where $n_{step} = n_{batch}$ when comparing performance per function evaluation, and $n_{step} = 1$ when comparing by infill iteration. Performance is sampled $n_{samples}$ times for the same configuration to correct for randomness.

For a given test problem, the best performing algorithm configuration has rank 1; higher ranks indicate lower performance. Similarly-performing configurations have the same rank, as tested by an independent two-sample t-test as implemented by Scipy's ttest_ind_from_stats[9] function. The algorithm for determining rank is listed in Algorithm 1. The best performing algorithm configuration is then selected by counting ranks over multiple test problems: the best performing configuration is the one with the highest proportion of rank 1 within the set of configurations with highest proportion of rank ≤ 2.

References

Crawley, E., Cameron, B., Selva, D.: System Architecture: Strategy and Product

[9] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_ind_from_stats.html

Algorithm 1 Determine performance rank R_i for each algorithm configuration C_i given performance measure p_i with standard deviation σ_i.

Require: C, p, σ, $n_{samples}$, $perfMin$
Ensure: R
$R \leftarrow zeros$ // Initialize ranks to 0 (unevaluated)
if $perfMin$ **then** // Get initial best performing configuration
    $i_{comp} \leftarrow \arg\min p$
else
    $i_{comp} \leftarrow \arg\max p$
end if
$R_{i_{comp}} \leftarrow 1$ // Set initial best performing configuration to rank 1
while $any(R = 0)$ **do** // Loop while there are unevaluated ranks
    $i_{uneval} \leftarrow \arg R = 0$ // Get unevaluated ranks
    if $perfMin$ **then** // Get best performing unevaluated configuration
        $j_{comp} \leftarrow \arg\min p(i_{uneval})$
    else
        $j_{comp} \leftarrow \arg\max p(i_{uneval})$
    end if
    $p_{same} \leftarrow tTestInd(p_{i_{comp}}, \sigma_{i_{comp}}, n_{samples}, p_{j_{comp}}, \sigma_{j_{comp}}, n_{samples})$
    if $p_{same} \leq 10\%$ **then** // If probability of being the same is low
        $R_{j_{comp}} \leftarrow R_{i_{comp}} + 1$ // Assign higher rank to compared configuration
        $i_{comp} \leftarrow j_{comp}$ // Update reference configuration
    else
        $R_{j_{comp}} \leftarrow R_{i_{comp}}$ // Assign same rank
    end if
end while

Development for Complex Systems. Pearson Education, England (2015). https://doi.org/10.1007/978-1-4020-4399-4

Chan, A., Pires, A.F., Polacsek, T., Roussel, S.: The aircraft and its manufacturing system: From early requirements to global design. In: International Conference on Advanced Information Systems Engineering (2022). https://doi.org/10.1007/978-3-031-07472-1_10

Iacobucci, J.V.: Rapid architecture alternative modeling (raam): a framework for capability-based analysis of system of systems architectures. PhD thesis, Georgia Institute of Technology (2012)

McDermott, T.A., Folds, D.J., Hallo, L.: Addressing cognitive bias in systems engineering teams. In: 30th Annual INCOSE International Symposium, Virtual Event (2020). https://doi.org/10.1002/j.2334-5837.2020.00721.x

Judt, D.M., Lawson, C.P.: Development of an automated aircraft subsystem architecture generation and analysis tool. Engineering Computations **33**(5), 1327–1352

(2016) https://doi.org/10.1108/EC-02-2014-0033

Bussemaker, J.H., Bartoli, N., Lefebvre, T., Ciampa, P.D., Nagel, B.: Effectiveness of surrogate-based optimization algorithms for system architecture optimization. In: AIAA AVIATION 2021 FORUM, Virtual Event (2021). https://doi.org/10.2514/6.2021-3095

Bussemaker, J.H., Ciampa, P.D.: MBSE in architecture design space exploration. In: Madni, A.M., Augustine, N., Sievers, M. (eds.) Handbook of Model-Based Systems Engineering. Springer, Switzerland (2022). https://doi.org/10.1007/978-3-030-27486-3_36-1

Bussemaker, J.H., Boggero, L., Ciampa, P.D.: From system architecting to system design and optimization: A link between MBSE and MDAO. In: 32nd Annual INCOSE International Symposium, Detroit, MI, USA (2022). https://doi.org/10.1002/iis2.12935

Martins, J.R.R.A., Ning, A.: Engineering Design Optimization. Cambridge University Press, Cambridge (2022). https://mdobook.github.io/

Frank, C.P., Marlier, R., Pinon-Fischer, O.J., Mavris, D.N.: An evolutionary multi-architecture multi-objective optimization algorithm for design space exploration. In: 57th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, Reston, Virginia, pp. 1–19 (2016). https://doi.org/10.2514/6.2016-0414

Saves, P., Bartoli, N., Diouane, Y., Lefebvre, T., Morlier, J., David, C., , Nguyen Van, E., Defoort, S.: Constrained bayesian optimization over mixed categorical variables, with application to aircraft design. In: AeroBest 2021 (2021). https://hal.science/hal-03346341v1/file/DTIS21090postprint.pdf

Saves, P., Diouane, Y., Bartoli, N., Lefebvre, T., Morlier, J.: A mixed-categorical correlation kernel for gaussian process. Neurocomputing **550**, 126472 (2023) https://doi.org/10.1016/j.neucom.2023.126472

Simmons, W.L.: A framework for decision support in systems architecting. PhD thesis, Massachusetts Institute of Technology (2008)

Pelamatti, J., Brevault, L., Balesdent, M., Talbi, E., Guerin, Y.: Bayesian optimization of variable-size design space problems. Optimization and Engineering (2020) https://doi.org/10.1007/s11081-020-09520-z

Feurer, M., Hutter, F.: Hyperparameter optimization. In: Automated Machine Learning, pp. 3–33. Springer, Switzerland (2019). https://doi.org/10.1007/978-3-030-05318-5_1

Armstrong, M., Tenorio, C., Garcia, E., Mavris, D.: Function based architecture design space definition and exploration. In: 26th Congress of International Council of the

Aeronautical Sciences, Anchorage, Alaska, USA (2008). https://doi.org/10.2514/6.2008-8928

Zaefferer, M., Horn, D.: A first analysis of kernels for kriging-based optimization in hierarchical search spaces. In: Parallel Problem Solving from Nature, PPSN XI vol. 1, pp. 399–410. Springer, Berlin, Heidelberg (2018). https://doi.org/10.1007/978-3-319-99259-4_32

Hutter, F., Osborne, M.A.: A kernel for hierarchical parameter spaces (2013) https://doi.org/10.48550/arXiv.1310.5738

Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B.: Algorithms for hyper-parameter optimization. In: Advances in Neural Information Processing Systems 24, Granada, Spain (2011)

Jenatton, R., Archambeau, C., González, J., Seeger, M.: Bayesian optimization with tree-structured dependencies. In: Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia (2017)

Apaza, G., Selva, D.: Automatic composition of encoding scheme and search operators in system architecture optimization. In: 41st Computers and Information in Engineering Conference (CIE). American Society of Mechanical Engineers, Virtual (2021). https://doi.org/10.1115/detc2021-71399

Abdelkhalik, O.: Hidden genes genetic optimization for variable-size design space problems. Journal of Optimization Theory and Applications **156**(2), 450–468 (2012) https://doi.org/10.1007/s10957-012-0122-6

Talbi, P.E.-G.: Metaheuristics for (variable-size) mixed optimization problems: A unified taxonomy and survey (2024) https://doi.org/10.48550/ARXIV.2401.03880

Levesque, J.-C., Durand, A., Gagne, C., Sabourin, R.: Bayesian optimization for conditional hyperparameter spaces. In: 2017 International Joint Conference on Neural Networks (IJCNN). IEEE, Anchorage, Alaska, USA (2017). https://doi.org/10.1109/ijcnn.2017.7965867

Selva, D.: Rule-based system architecting of earth observation satellite systems. PhD thesis, Massachusetts Institute of Technology, Dept. of Aeronautics and Astronautics (2012). http://hdl.handle.net/1721.1/76089

Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., Wasowski, A.: Cool features and tough decisions. In: 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

Weilkiens, T., Lamm, J.G., Roth, S.: Model-Based System Architecture. Wiley John and Sons, Hoboken, NJ, USA (2015). https://doi.org/10.1002/9781119051930

Le Digabel, S., Wild, S.M.: A taxonomy of constraints in black-box simulation-based optimization. Optimization and Engineering (2023) https://doi.org/10.1007/s11081-023-09839-3

Ying, C., Klein, A., Christiansen, E., Real, E., Murphy, K., Hutter, F.: NAS-bench-101: Towards reproducible neural architecture search. In: Proceedings of the 36th International Conference on Machine Learning. PMLR, Long Beach, CA, USA (2019)

Salcedo-Sanz, S.: A survey of repair methods used as constraint handling techniques in evolutionary algorithms. Computer Science Review **3**(3), 175–192 (2009) https://doi.org/10.1016/j.cosrev.2009.07.001

Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: A literature review. Information Systems **35**(6), 615–636 (2010) https://doi.org/10.1016/j.is.2010.01.001

Bussemaker, J.H., De Smedt, T., La Rocca, G., Ciampa, P.D., Nagel, B.: System architecture optimization: An open source multidisciplinary aircraft jet engine architecting problem. In: AIAA AVIATION 2021 FORUM, Virtual Event (2021). https://doi.org/10.2514/6.2021-3078

Sobieszczanski-Sobieski, J., Morris, A., van Tooren, M.J.L.: Multidisciplinary Design Optimization Supported by Knowledge Based Engineering, pp. 1–378. John Wiley & Sons, Ltd, West Sussex, UK (2015). https://doi.org/10.1002/9781118897072

Jones, D.R., Schonlau, M., Welch, W.J.: Efficient global optimization of expensive black-box functions. Journal of Global Optimization **13**, 455–492 (1998) https://doi.org/10.1023/A:1008306431147

Miettinen, K.: Nonlinear Multiobjective Optimization. Springer, USA (1998). https://doi.org/10.1007/978-1-4615-5563-6

Rudenko, O., Schoenauer, M.: A steady performance stopping criterion for Pareto-based evolutionary algorithms. In: 6th International Multi-Objective Programming and Goal Programming Conference, Hammamet, Tunisia (2004)

Priem, R., Bartoli, N., Diouane, Y.: On the Use of Upper Trust Bounds in Constrained Bayesian Optimization Infill Criteria. In: AIAA Aviation 2019 Forum, pp. 1–10. American Institute of Aeronautics and Astronautics, Reston, Virginia (2019). https://doi.org/10.2514/6.2019-2986

Forrester, A.I.J., Sóbester, A., Keane, A.J.: Optimization with missing data. Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences **462**(2067), 935–945 (2006) https://doi.org/10.1098/rspa.2005.1608

Müller, J., Day, M.: Surrogate optimization of computationally expensive black-box

problems with hidden constraints. INFORMS Journal on Computing **31**(4), 689–702 (2019) https://doi.org/10.1287/ijoc.2018.0864

Krengel, M.D., Hepperle, M.: Effects of wing elasticity and basic load alleviation on conceptual aircraft designs. In: AIAA SCITECH 2022 Forum (2022). https://doi.org/10.2514/6.2022-0126

Yang, L., Shami, A.: On hyperparameter optimization of machine learning algorithms: Theory and practice. Neurocomputing **415**, 295–316 (2020) https://doi.org/10.1016/j.neucom.2020.07.061

Jones, D.R., Martins, J.R.R.A.: The DIRECT algorithm: 25 years later. Journal of Global Optimization **79**(3), 521–566 (2020) https://doi.org/10.1007/s10898-020-00952-6

Locatelli, M., Schoen, F.: (Global) optimization: Historical notes and recent developments. EURO Journal on Computational Optimization **9**, 100012 (2021) https://doi.org/10.1016/j.ejco.2021.100012

Glover, F., Kochenberger, G.: Handbook of Metaheuristics vol. 57, pp. 457–474474 (2003). https://doi.org/10.1007/b101874

Petrowski, A., Ben-Hamida, S.: Evolutionary Algorithms, p. 256. John Wiley & Sons, London, UK (2017)

Hamano, R., Saito, S., Nomura, M., Shirakawa, S.: CMA-ES with margin. In: Proceedings of the Genetic and Evolutionary Computation Conference. ACM, Boston, US (2022). https://doi.org/10.1145/3512290.3528827

Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation **6**(2), 182–197 (2002) https://doi.org/10.1109/4235.996017

Nyew, H.M., Abdelkhalik, O., Onder, N.: Structured-chromosome evolutionary algorithms for variable-size autonomous interplanetary trajectory planning optimization. Journal of Aerospace Information Systems **12**(3), 314–328 (2015) https://doi.org/10.2514/1.i010272

Gratton, S., Vicente, L.N.: A merit function approach for direct search. SIAM Journal on Optimization **24**(4), 1980–1998 (2014) https://doi.org/10.1137/130917661

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

Buonanno, M.A.: A method for aircraft concept exploration using multicriteria interactive genetic algorithms. PhD thesis, Georgia Institute of Technology (August

2005)

Pate, D.J., Patterson, M.D., German, B.J.: Optimizing Families of Reconfigurable Aircraft for Multiple Missions. Journal of Aircraft **49**(6), 1988–2000 (2012) https://doi.org/10.2514/1.C031667

Chugh, T., Sindhya, K., Hakanen, J., Miettinen, K.: A survey on handling computationally expensive multiobjective optimization problems with evolutionary algorithms. Soft Computing **23**(9), 3137–3166 (2019) https://doi.org/10.1007/s00500-017-2965-0

Regis, R.G., Shoemaker, C.A.: Combining radial basis function surrogates and dynamic coordinate search in high-dimensional expensive black-box optimization. Engineering Optimization **45**(5), 529–555 (2013) https://doi.org/10.1080/0305215x.2012.687731

Garnett, R.: Bayesian Optimization. Cambridge University Press, Cambridge, UK (2023). https://doi.org/10.1017/9781108348973

Schonlau, M., Welch, W.J., Jones, D.R.: Global versus local search in constrained optimization of computer models. In: Lecture Notes - Monograph Series, pp. 11–25. Institute of Mathematical Statistics, online (1998). https://doi.org/10.1214/lnms/1215456182

Sasena, M.J.: Flexibility and Efficiency Enhancements for Constrained Global Design Optimization with Kriging Approximations. PhD thesis, University of Michigan (2002)

Knowles, J.: ParEGO: a hybrid algorithm with on-line landscape approximation for expensive multiobjective optimization problems. IEEE Transactions on Evolutionary Computation **10**(1), 50–66 (2006) https://doi.org/10.1109/TEVC.2005.851274

Rojas-Gonzalez, S., Van Nieuwenhuyse, I.: A survey on Kriging-based infill algorithms for multiobjective simulation optimization. Computers & Operations Research **116**, 104869 (2019) https://doi.org/10.1016/j.cor.2019.104869

Garrido-Merchán, E.C., Hernández-Lobato, D.: Dealing with categorical and integer-valued variables in bayesian optimization with gaussian processes. Neurocomputing **380**, 20–35 (2020) https://doi.org/10.1016/j.neucom.2019.11.004

Daulton, S., Wan, X., Eriksson, D., Balandat, M., Osborne, M.A., Bakshy, E.: Bayesian optimization over discrete and mixed spaces via probabilistic reparameterization (2022) https://doi.org/10.48550/ARXIV.2210.10199

Pelamatti, J., Brevault, L., Balesdent, M., Talbi, E., Guerin, Y.: Efficient global optimization of constrained mixed variable problems. Journal of Global Optimization

73(3), 583–613 (2019) https://doi.org/10.1007/s10898-018-0715-1

Zuniga, M.M., Sinoquet, D.: Global optimization for mixed categorical-continuous variables based on gaussian process models with a randomized categorical space exploration step. INFOR: Information Systems and Operational Research **58**(2), 310–341 (2020) https://doi.org/10.1080/03155986.2020.1730677

Dreczkowski, K., Grosnit, A., Ammar, H.B.: Framework and benchmarks for combinatorial and mixed-variable bayesian optimization (2023) https://doi.org/10.48550/ARXIV.2306.09803

Audet, C., Hallé-Hannan, E., Le Digabel, S.: A general mathematical framework for constrained mixed-variable blackbox optimization problems with meta and categorical variables. Operations Research Forum **4**, 1–37 (2023)

Horn, D., Stork, J., Schüßler, N., Zaefferer, M.: Surrogates for hierarchical search spaces. In: Proceedings of the Genetic and Evolutionary Computation Conference. ACM, Prague, CZ (2019). https://doi.org/10.1145/3321707.3321765

Lu, X., Gonzalez, J., Dai, Z., Lawrence, N.: Structured variationally auto-encoded optimization. In: Proceedings of the 35th International Conference on Machine Learning. PMLR, Stockholm, SE (2018)

Saves, P., Lafage, R., Bartoli, N., Diouane, Y., Bussemaker, J.H., Lefebvre, T., Hwang, J.T., Morlier, J., Martins, J.R.R.A.: SMT 2.0: A surrogate modeling toolbox with a focus on hierarchical and mixed variables gaussian processes. Advances in Engineering Software **188**, 103571 (2024) https://doi.org/10.1016/j.advengsoft.2023.103571

Bouhlel, M.A., Bartoli, N., Regis, R.G., Otsmane, A., Morlier, J.: Efficient global optimization for high-dimensional constrained problems by using the kriging models combined with the partial least squares method. Engineering Optimization **50**(12), 2038–2053 (2018) https://doi.org/10.1080/0305215x.2017.1419344

Priem, R., Bartoli, N., Diouane, Y., Dubreuil, S., Saves, P.: High-dimensional efficient global optimization using both random and supervised embeddings. In: AIAA AVIATION 2023 Forum. American Institute of Aeronautics and Astronautics, San Diego, CA, USA (2023). https://doi.org/10.2514/6.2023-4448

Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Lecture Notes in Computer Science, pp. 507–523. Springer, Berlin Heidelberg (2011). https://doi.org/10.1007/978-3-642-25566-3_40

Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Deng, D., Benjamins, C., Ruhkopf, T., Sass, R., Hutter, F.: SMAC3: A versatile Bayesian optimization package for hyperparameter optimization. Journal of Machine Learning Research

23(54), 1–9 (2022)

Ozaki, Y., Tanigaki, Y., Watanabe, S., Nomura, M., Onishi, M.: Multiobjective tree-structured parzen estimator. Journal of Artificial Intelligence Research **73**, 1209–1250 (2022) https://doi.org/10.1613/jair.1.13188

Eggensperger, K., Hutter, F., Hoos, H., Leyton-Brown, K.: Efficient benchmarking of hyperparameter optimizers via surrogates. Proceedings of the AAAI Conference on Artificial Intelligence **29**(1) (2015) https://doi.org/10.1609/aaai.v29i1.9375

Gamot, J., Balesdent, M., Tremolet, A., Wuilbercq, R., Melab, N., Talbi, E.-G.: Hidden-variables genetic algorithm for variable-size design space optimal layout problems with application to aerospace vehicles. Engineering Applications of Artificial Intelligence **121**, 105941 (2023)

Greenhill, S., Rana, S., Gupta, S., Vellanki, P., Venkatesh, S.: Bayesian optimization for adaptive experimental design: A review. IEEE Access **8**, 13937–13948 (2020) https://doi.org/10.1109/access.2020.2966228

Calandra, R., Peters, J., Rasmussen, C.E., Deisenroth, M.P.: Manifold gaussian processes for regression. In: 2016 International Joint Conference on Neural Networks (IJCNN). IEEE, Vancouver, CA (2016). https://doi.org/10.1109/ijcnn.2016.7727626

Bouhlel, M.A., Bartoli, N., Otsmane, A., Morlier, J.: Improving Kriging surrogates of high-dimensional design models by Partial Least Squares dimension reduction. Structural and Multidisciplinary Optimization **53**(5), 935–952 (2016) https://doi.org/10.1007/s00158-015-1395-9

Saves, P., Bartoli, N., Diouane, Y., Lefebvre, T., Morlier, J., David, C., Van, E.N., Defoort, S.: Multidisciplinary design optimization with mixed categorical variables for aircraft design. In: AIAA SCITECH 2022 Forum. American Institute of Aeronautics and Astronautics, San Diego, CA, USA (2022). https://doi.org/10.2514/6.2022-0082

Charayron, R., Lefebvre, T., Bartoli, N., Morlier, J.: Multi-fidelity Bayesian optimization strategy applied to overall drone design. In: AIAA SCITECH 2023 Forum. American Institute of Aeronautics and Astronautics, National Harbor, MD, USA (2023). https://doi.org/10.2514/6.2023-2366

Bouhlel, M.A., Hwang, J.T., Bartoli, N., Lafage, R., Morlier, J., Martins, J.R.R.A.: A Python surrogate modeling framework with derivatives. Advances in Engineering Software **135**, 102662 (2019) https://doi.org/10.1016/j.advengsoft.2019.03.005

Lyu, W., Yang, F., Yan, C., Zhou, D., Zeng, X.: Batch Bayesian optimization via multi-objective acquisition ensemble for automated analog circuit design. In: Proceedings of the 35th International Conference on Machine Learning. PMLR, Stockholm, SE

(2018)

Cowen-Rivers, A.I., Lyu, W., Tutunov, R., Wang, Z., Grosnit, A., Griffiths, R.R., Maraval, A.M., Jianye, H., Wang, J., Peters, J., Ammar, H.B.: HEBO: Pushing the limits of sample-efficient hyperparameter optimisation (2020) https://doi.org/10.48550/ARXIV.2012.03826

Ginsbourger, D., Riche, R.L., Carraro, L.: Kriging is well-suited to parallelize optimization. In: Computational Intelligence in Expensive Optimization Problems, pp. 131–162. Springer, Berlin Heidelberg (2010). https://doi.org/10.1007/978-3-642-10701-6_6

Cox, D.D., John, S.: A statistical method for global optimization. In: 1992 IEEE International Conference on Systems, Man, and Cybernetics. IEEE, Chicago, IL, USA (1992). https://doi.org/10.1109/icsmc.1992.271617

Hawe, G.I., Sykulski, J.K.: An enhanced probability of improvement utility function for locating pareto optimal solutions. 16th Conference on the Computation of Electromagnetic Fields, COMPUMAG, Aachen, Germany **3**, 965–966 (2007)

Rahat, A.A.M., Everson, R.M., Fieldsend, J.E.: Alternative infill strategies for expensive multi-objective optimisation. In: Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO '17, pp. 873–880. ACM Press, New York, USA (2017). https://doi.org/10.1145/3071178.3071276

Sohst, M., Afonso, F., Suleman, A.: Surrogate-based optimization based on the probability of feasibility. Structural and Multidisciplinary Optimization **65**(1) (2021) https://doi.org/10.1007/s00158-021-03134-4

Bussemaker, J.H.: SBArchOpt: Surrogate-based architecture optimization. Journal of Open Source Software **8**(89), 5564 (2023) https://doi.org/10.21105/joss.05564

Blank, J., Deb, K.: Pymoo: Multi-objective optimization in python. IEEE Access **8**, 89497–89509 (2020) https://doi.org/10.1109/access.2020.2990567

Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Marben, J., Müller, P., Hutter, F.: BOAH: A tool suite for multi-fidelity Bayesian optimization & analysis of hyperparameters (2019) https://doi.org/10.48550/ARXIV.1908.06756

Balandat, M., Karrer, B., Jiang, D.R., Daulton, S., Letham, B., Wilson, A.G., Bakshy, E.: BoTorch: A framework for efficient monte-carlo Bayesian optimization. Advances in Neural Information Processing Systems 33 (2019) https://doi.org/10.48550/ARXIV.1910.06403

Picheny, V., Berkeley, J., Moss, H.B., Stojic, H., Granta, U., Ober, S.W., Artemev, A., Ghani, K., Goodall, A., Paleyes, A., Vakili, S., Pascual-Diaz, S., Markou, S., Qing, J., Loka, N.R.B.S., Couckuyt, I.: Trieste: Efficiently exploring the depths of black-box

functions with TensorFlow (2023) https://doi.org/10.48550/ARXIV.2302.08436

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

Bekemeyer, P., Bertram, A., Hines Chaves, D.A., Dias Ribeiro, M., Garbo, A., Kiener, A., Sabater, C., Stradtner, M., Wassing, S., Widhalm, M., Goertz, S., Jaeckel, F., Hoppe, R., Hoffmann, N.: Data-driven aerodynamic modeling using the dlr smarty toolbox. In: AIAA AVIATION 2022 Forum. American Institute of Aeronautics and Astronautics, Chicago, USA (2022). https://doi.org/10.2514/6.2022-3899

García Sánchez, R.: Adaptation of an MDO platform for system architecture optimization. mathesis, Delft University of Technology, Delft, NL (January 2024)

Saves, P., Diouane, Y., Bartoli, N., Lefebvre, T., Morlier, J.: A general square exponential kernel to handle mixed-categorical variables for gaussian process. In: AIAA AVIATION 2022 Forum. American Institute of Aeronautics and Astronautics, Chicago, IL, USA (2022). https://doi.org/10.2514/6.2022-3870

Renardy, M., Joslyn, L.R., Millar, J.A., Kirschner, D.E.: To sobol or not to sobol? the effects of sampling schemes in systems biology applications. Mathematical Biosciences **337**, 108593 (2021) https://doi.org/10.1016/j.mbs.2021.108593

Kaltenecker, C., Grebhahn, A., Siegmund, N., Guo, J., Apel, S.: Distance-based sampling of software configuration spaces. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, Montreal, CA (2019). https://doi.org/10.1109/icse.2019.00112

Collette, Y., Hansen, N., Pujol, G., Aponte, D.S., Riche, R.L.: Object-Oriented Programming of Optimizers – Examples in Scilab. In: Breitkopf, P., Coelho, R.F. (eds.) Multidisciplinary Design Optimization in Computational Mechanics, pp. 499–538. Wiley, London, UK (2013). https://doi.org/10.1002/9781118600153.ch14

Li, Y., Shen, Y., Zhang, W., Chen, Y., Jiang, H., Liu, M., Jiang, J., Gao, J., Wu, W., Yang, Z., Zhang, C., Cui, B.: OpenBox: A generalized black-box optimization service. In: Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. ACM, New York, USA (2021). https://doi.org/10.1145/3447548.3467061

Mavris, D., de Tenorio, C., Armstrong, M.: Methodology for aircraft system architecture definition. In: 46th AIAA Aerospace Sciences Meeting and Exhibit, pp. 1–14. American Institute of Aeronautics and Astronautics, Reston, Virigina (2008). https://doi.org/10.2514/6.2008-149

Gedell, S., Johannesson, H.: Design rationale and system description aspects in

product platform design: Focusing reuse in the design lifecycle phase. Concurrent Engineering **21**(1), 39–53 (2012) https://doi.org/10.1177/1063293x12469216

Zhang, Q., Han, Z., Yang, F., Zhang, Y., Liu, Z., Yang, M., Zhou, L.: Retiarii: A deep learning Exploratory-Training framework. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pp. 919–936. USENIX Association, Virtual (2020). https://www.usenix.org/conference/osdi20/presentation/zhang-quanlu

Elsken, T., Metzen, J.H., Hutter, F.: Neural architecture search: A survey. Journal of Machine Learning Research **20**(55), 1–21 (2019)

Gray, J.S., Hwang, J.T., Martins, J.R.R.A., Moore, K.T., Naylor, B.A.: OpenMDAO: an open-source framework for multidisciplinary design, analysis, and optimization. Structural and Multidisciplinary Optimization **59**(4), 1075–1104 (2019) https://doi.org/10.1007/s00158-019-02211-z

Hendricks, E.S., Gray, J.S.: pyCycle: A tool for efficient optimization of gas turbine engine cycles. Aerospace **6**(8), 87 (2019) https://doi.org/10.3390/aerospace6080087

Donelli, G., Ciampa, P.D., Mello, J.M.G., Odaguil, F.I.K., Cuco, A.P.C., Laan, T.: A value-driven concurrent approach for aircraft design-manufacturing-supply chain. Production and Manufacturing Research **11**(1) (2023) https://doi.org/10.1080/21693277.2023.2279709