System Architecture Optimization

Function-Based Modeling,
Optimization Algorithms,
and Multidisciplinary Evaluation

Jasper H. Bussemaker

SYSTEM ARCHITECTURE OPTIMIZATION

FUNCTION-BASED MODELING, OPTIMIZATION ALGORITHMS, AND MULTIDISCIPLINARY EVALUATION

Dissertation

for the purpose of obtaining the degree of doctor
at the Delft University of Technology
by the authority of the Rector Magnificus Prof.dr.ir. T.H.J.J. van der Hagen;
Chair of the Board for Doctorates
to be defended publicly on Friday 4 July 2025 at 15:00 o'clock

by

Jasper Herm BUSSEMAKER

Master of Science in Aerospace Engineering,
Delft University of Technology, the Netherlands,
born in Amsterdam, the Netherlands.

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus	chairperson
Dr. ir. G. La Rocca	Delft University of Technology, *promotor*
Dr. N. Bartoli	ISAE-SUPAERO, France, *promotor*
Independent members:	
Prof. dr. E.K.A. Gill	Delft University of Technology
Prof. dr. ir. M. Langelaar	Delft University of Technology
Prof. Dr.-Ing. J. Abulawi	HAW-Hamburg, Germany
Dr. D. Selva Valero	Texas A&M University, USA
Dr. C.D. Jouannet	Linköping University / Saab, Sweden

This research has been carried out at the Institute of System Architectures in Aeronautics of the German Aerospace Center (DLR) in Hamburg, Germany, in the context of an agreement on joint doctoral supervision between Delft University of Technology and ISAE-SUPAERO in Toulouse, France.

This research was partially funded by the AGILE 4.0 project from the EU Horizon 2020 Programme under grant agreement n° 815122, and the COLOSSUS project from the EU Horizon Europe Programme under grant agreement n° 101097120.

Keywords: system architecture, optimization, systems engineering, MBSE, MDAO

Printed by: Ipskamp Printing

ISBN 978-94-6518-083-0

An electronic version of this dissertation is available at
http://repository.tudelft.nl/.

to Carlo and Melissa

"Design must reflect the practical and aesthetic in business,
but above all ... good design must primarily serve people."

- Thomas J. Watson

"We are drowning in information, while starving for wisdom.
The world henceforth will be run by synthesizers, people able
to put together the right information at the right time, think
critically about it, and make important choices wisely."

- Edward O. Wilson

Contents

Summary

Designing complex systems that are increasingly subject to sustainability, economic, and safety constraints requires the integration of disruptive technologies, while considering the entire system life-cycle and weighting conflicting stakeholder needs. The **system architecture** describes what *functions* the system performs in order to meet the design goals, and how the *components* of the system collaborate to fulfill these functions. Architectural choices are taken early in the systems engineering process, and greatly influence to which extent design goals are achieved by the system.

Due to the combinatorial nature of architectural choices, the architecture design space, which represents the set of all possible architectures for a given design problem, can be extremely large. Additionally, integrating innovative technologies requires the application of multidisciplinary, simulation-based evaluation, due to a lack of historical data to base decisions on. These two challenges drive the need for a design methodology that allows exploring the architecture design space without the need to evaluate *all* possible architectures, as that would be infeasible.

System Architecture Optimization (SAO) does just that: it combines numerical optimization algorithms for automatically searching the architecture design space (the "architecture generator") with multidisciplinary, simulation-based evaluation of architecture candidates (the "architecture evaluator").

The **architecture generator** consists of two elements:

- A way for the system architect to **model the architecture design space** and formulate the optimization problem.

  The architecture design space model should be able to represent function decomposition, function-to-component allocation, function and component characterization and specialization, as well as component connection choices. Once the architecture optimization problem has been defined, it should then be automatically encoded into a formalized numerical optimization problem, and the generated design vectors should be decoded into architecture instances.

  Currently, no combination of modeling language and encoding and decoding algorithms exists that supports all of the above.

- **Optimization algorithms** that can solve the formulated optimization problem.

  SAO problems are challenging, because in general they feature mixed-discrete design variables, design variable hierarchy, black-box and expensive evaluation functions, multiple conflicting objectives, design constraints, and hidden constraints. Design variable hierarchy comes from activation relationships (variables determining whether other variables are active) and from value constraints (variables restricting the available options of other variables). Evolutionary algorithms and Bayesian optimization algorithms are global optimization algorithms with the potential to solve such problems.

However, existing sampling algorithms[1] *do not explicitly consider the hierarchical nature of the design space, thereby potentially over- or under-sampling certain regions in the design space. Additionally, it is not clear how the correction algorithm*[2] *influences optimization performance. Finally, Bayesian optimization algorithms have not been demonstrated for the combination of all SAO problem challenges.*

The **architecture evaluator** calculates the performance of a given architecture instance. The multidisciplinary nature of systems engineering and the need for simulation-based evaluation calls for the application of Multidisciplinary Design Analysis and Optimization (MDAO). Collaborative MDAO extends this by supporting the coupling of diverse analysis tools that are developed, managed, and executed by distributed teams and/or organizations in large systems engineering project.

Currently, however, collaborative MDAO workflows are static whereas their behavior should be flexible in order to be able to process all generated architecture instances. Additionally, all relevant architecture information should be communicated to the MDAO workflow, not only numerical parameter values as is currently done. Finally, the architecture generator should be integrated in the same computational environment as where the MDAO workflow is executed.

This work contributes towards the practical application of SAO by addressing the science gaps presented above.

Optimization Algorithms for SAO

In the first part, efficient global **optimization algorithms for SAO** are made available by:

- developing a hierarchical sampling algorithm that prevents over- or under-sampling regions in the hierarchical design space;
- showing that problem-specific correction is sufficient: problem-agnostic correction algorithms do not significantly improve optimization performance;
- integrating information about the hierarchical design space into the optimization algorithms, to support generating valid design vectors and to leverage activeness information for sampling and surrogate model creation (for Bayesian optimization); and
- developing a strategy for handling hidden constraints (due to simulation failures) for Bayesian optimization by predicting the Probability of Viability of selected infill points.

Above developments are demonstrated using three benchmark problems and a jet engine architecture optimization problem featuring hidden constraints. It is shown that Bayesian optimization can solve SAO problems with up to 92% less function evaluations compared to evolutionary algorithms. In practice this means that optimization results

[1] A sampling algorithm generates the initial set of architectures to initialize global optimization algorithms.
[2] A correction algorithm ensures that value constraints are satisfied.

are available sooner, or that a larger design space can be explored within the same time. Optimization algorithms and test problems are available open-source[3].

MODELING SAO PROBLEMS

In the second part, a methodology for **modeling SAO problems** is developed, consisting of:

- the Design Space Graph (DSG) for modeling hierarchical selection and connection choices, for automatically encoding these choices into design variables, and for automatically decoding design vectors into graph instances;
- the Architecture Design Space Graph (ADSG), extending the DSG with node types and rules specific to system architecting, such as functions, components, and ports; and
- ADORE, providing a web-based graphical user interface for creating and inspecting the ADSG, and providing various interfaces for connecting to evaluation code and optimization algorithms.

The DSG is available open-source[4].

A bottom-up function-based process is defined to support engineers in applying ADORE for modeling SAO problems. It is shown that this process results in a more natural approach compared to existing top-down processes. The methodology is demonstrated by a hybrid-electric propulsion SAO problem. An additional investigation using three test problems shows that optimization problems defined and encoded by ADORE perform as well as or better than manually-formulated optimization problems.

COLLABORATIVE MDAO FOR SAO

In the third part, **collaborative MDAO** is extended for use in SAO problems:

- The behavior of the collaborative MDAO workflow is dynamically modified during runtime to be appropriate for a given architecture instance, such as by modifying data connections between tools, modifying the number of repeated executions of tools, or by dynamically including or excluding tools.
- Architecture data is propagated from ADORE to the central data schema[5] of the MDAO workflow using a rule-based translation mechanism.
- ADORE is integrated in the computational environment where the collaborative MDAO workflow is executed by applying a modified ask-tell pattern.

Collaborative MDAO for SAO is demonstrated by the design of a multi-stage rocket.

[3]SBArchOpt: https://sbarchopt.readthedocs.io/
[4]ADSG-Core: https://adsg-core.readthedocs.io/
[5]A central data schema defines the common language for data exchange between disciplinary analysis tools in an MDAO problem.

To summarize, by combining

- **efficient global optimization algorithms** that are modified for application to SAO problems, resulting in a **92% reduction in function evaluations** compared to existing algorithms,
- a **function-based methodology for modeling** architectural design spaces, supporting engineers in formulating SAO problems **without requiring expertise in numerical optimization**, and
- **collaborative MDAO** extended to be used for evaluating architecture instances, enabling the application of SAO in **large cross-organizational systems engineering projects**,

engineers can use SAO to automatically explore large architectural design spaces for finding the best architecture instance for their design problem.

Samenvatting

Het ontwerpen van complexe systemen die in toenemende mate onderhevig zijn aan voorwaarden op het gebied van duurzaamheid, economie en veiligheid vraagt om de integratie van innovatieve technologieën. Hierbij moet rekening worden gehouden met de gehele levenscyclus van het systeem en de tegenstrijdige doelen van belanghebbenden. De **systeemarchitectuur** beschrijft welke *functies* het systeem uitvoert om aan de ontwerpdoelen te voldoen en hoe de *componenten* van het systeem samenwerken om deze functies te vervullen. De keuzes voor de architectuur worden vroeg in het systeemontwerpproces gemaakt en hebben grote invloed op de mate waarin de ontwerpdoelen door het systeem worden bereikt.

Door de combinatorische aard van architecturale keuzes kan de architectuurontwerpruimte, die de verzameling van alle mogelijke architecturen voor een bepaald ontwerpprobleem vertegenwoordigt, extreem groot zijn. Bovendien vereist de integratie van innovatieve technologieën de toepassing van multidisciplinaire, op simulatie gebaseerde evaluatie, vanwege een gebrek aan historische gegevens om beslissingen op te baseren. Deze twee uitdagingen zorgen voor de behoefte aan een ontwerpmethodologie die het mogelijk maakt om de architectuurontwerpruimte te verkennen zonder *alle* mogelijke architecturen te evalueren, aangezien dat niet haalbaar is.

Systeemarchitectuuroptimalisatie (SAO) doet precies dat: het combineert numerieke optimalisatiealgoritmes om automatisch de architectuurontwerpruimte te doorzoeken (de “architectuurgenerator”) met multidisciplinaire, op simulatie gebaseerde evaluatie van architectuurkandidaten (de “architectuurevaluator”).

De **architectuurgenerator** bestaat uit twee elementen:

- Een manier voor de systeemarchitect om **de architectuurontwerpruimte te modelleren** en het optimalisatieprobleem te formuleren.

  Het model van de architectuurontwerpruimte moet keuzes voor functiedecompositie, functietoewijzing aan componenten, de karakterisering en specialisatie van functies en componenten, en verbindingen tussen componenten kunnen representeren. Zodra het architectuuroptimalisatieprobleem is gedefinieerd, moet het automatisch kunnen worden gecodeerd als een geformaliseerd numeriek optimalisatieprobleem en moeten de gegenereerde ontwerpvectoren kunnen worden gedecodeerd als architectuurinstanties.

  Momenteel bestaat er geen combinatie van modelleertaal en coderings- en decoderingsalgoritmes die al het bovenstaande ondersteunt.

- **Optimalisatiealgoritmes** die het geformuleerde optimalisatieprobleem kunnen oplossen.

  SAO-problemen zijn uitdagend, omdat ze in het algemeen gekenmerkt worden door gemengd-discrete ontwerpvariabelen, hiërarchie van ontwerpvariabelen,

black-box en dure evaluatiefuncties, meerdere conflicterende doelfuncties, ontwerpbeperkingen en verborgen beperkingen. De hiërarchie van ontwerpvariabelen komt voort uit activeringsrelaties (variabelen die bepalen of andere variabelen actief zijn) en uit waardebeperkingen (variabelen die de beschikbare opties van andere variabelen beperken). Evolutionaire algoritmes en Bayesiaanse optimalisatiealgoritmes zijn globale optimalisatiealgoritmes die dergelijke problemen kunnen oplossen.

Huidige steekproefalgoritmes[6] *houden echter niet expliciet rekening met de hiërarchische aard van de ontwerpruimte, waardoor bepaalde gebieden in de ontwerpruimte mogelijk over- of onderbemonsterd worden. Bovendien is het niet duidelijk hoe het correctiealgoritme*[7] *de optimalisatieprestaties beïnvloedt. Tot slot is Bayesiaanse optimalisatie nog niet gedemonstreerd voor alle uitdagingen van SAO-problemen tegelijk.*

De **architectuurevaluator** berekent hoe goed een gegeven architectuurinstantie presteert. De multidisciplinaire aard van systeembouw en de behoefte aan evaluatie gebaseerd op simulatie maakt het nodig Multidisciplinaire Ontwerpanalyse en Optimalisatie (MDAO) toe te passen. Collaboratieve MDAO voegt daar aan toe dat het mogelijk wordt gemaakt om diverse analysemodellen aan elkaar te koppelen die worden ontwikkeld, beheerd en uitgevoerd door verspreid werkende teams en/of organisaties in grote systeembouwprojecten.

Momenteel zijn collaboratieve MDAO-processen echter statisch, terwijl hun gedrag flexibel zou moeten zijn om alle gegenereerde architectuurinstanties te kunnen verwerken. Bovendien moet alle relevante architectuurinformatie worden doorgegeven aan het MDAO-proces, niet alleen numerieke parameterwaarden zoals nu het geval is. Tot slot moet de architectuurgenerator in dezelfde rekenomgeving worden geïntegreerd als waar het MDAO-proces wordt uitgevoerd.

Door de hierboven beschreven open punten in mijn onderzoek aan te pakken, komt de toepassing van SAO in de praktijk dichterbij.

OPTIMALISATIEALGORITMES VOOR SAO

In het eerste deel worden efficiënte globale **optimalisatiealgoritmes voor SAO** toepasbaar gemaakt door:

- een hiërarchisch steekproefalgoritme te ontwikkelen dat voorkomt dat sommige gebieden in de ontwerpruimte over- of onderbemonsterd worden;
- aan te tonen dat probleemspecifieke correctie voldoende is: probleemagnostische correctiealgoritmes verbeteren de optimalisatieprestaties niet significant;
- het integreren van informatie over de hiërarchische ontwerpruimte in de optimalisatiealgoritmes, om het genereren van geldige ontwerpvectoren te ondersteunen en om informatie over activiteit te gebruiken voor het nemen van steekproeven en het creëren van surrogaatmodellen (voor Bayesiaanse optimalisatie); en

[6] Een steekproefalgoritme genereert de initiële reeks architecturen om globale optimalisatiealgoritmes te initialiseren.
[7] Een correctiealgoritme zorgt ervoor dat aan de waardebeperkingen wordt voldaan.

- het ontwikkelen van een strategie voor het omgaan met verborgen beperkingen (door simulatiefouten) voor Bayesiaanse optimalisatie door het voorspellen van de waarschijnlijkheid dat geselecteerde invulpunten te simuleren zijn.

Bovengenoemde ontwikkelingen worden gedemonstreerd aan de hand van drie testproblemen en de optimalisatie van een straalmotorarchitectuur met verborgen beperkingen. Het wordt aangetoond dat het Bayesiaanse optimalisatiealgoritme SAO-problemen kan oplossen met tot 92% minder functie-evaluaties in vergelijking met evolutionaire algoritmes. In de praktijk betekent dit dat optimalisatieresultaten sneller beschikbaar zijn, of dat een grotere ontwerpruimte binnen dezelfde tijd kan worden verkend. Optimalisatiealgoritmes en testproblemen zijn open-source beschikbaar[8].

MODELLERING VAN SAO-PROBLEMEN

In het tweede deel wordt een methodologie ontwikkeld voor **het modelleren van SAO-problemen**, bestaande uit:

- de Ontwerpruimtegraaf voor het modelleren van hiërarchische selectie- en verbindingskeuzes, voor het automatisch coderen van deze keuzes als ontwerpvariabelen, en voor het automatisch decoderen van ontwerpvectoren naar graafinstanties;
- de Architectuurontwerpruimtegraaf voegt aan de Ontwerpruimtegraaf knooptypes en regels toe die specifiek zijn voor systeemarchitectuur, zoals functies, componenten en poorten; en
- ADORE, dat een webgebaseerde grafische gebruikersomgeving biedt voor het creëren en inspecteren van de Architectuurontwerpruimtegraaf, en verschillende interfaces biedt voor het verbinden met evaluatiecode en optimalisatiealgoritmes.

De Ontwerpruimtegraaf is open-source beschikbaar[9].

Een bottom-up op functies gebaseerd proces wordt gedefinieerd om ingenieurs te ondersteunen bij het gebruik van ADORE voor het modelleren van SAO-problemen. Het wordt aangetoond dat dit proces resulteert in een natuurlijker aanpak in vergelijking met bestaande top-down processen. De methodologie wordt gedemonstreerd aan de hand van een SAO-probleem voor hybride-elektrische aandrijving. Een aanvullend onderzoek met drie testproblemen toont aan dat optimalisatieproblemen die door ADORE gedefinieerd en gecodeerd zijn, even goed of beter presteren dan handmatig geformuleerde optimalisatieproblemen.

COLLABORATIEVE MDAO VOOR SAO

In het derde deel wordt **collaboratieve MDAO** uitgebreid voor het gebruik in SAO-problemen:

- Het gedrag van het collaboratieve MDAO-proces wordt tijdens runtime dynamisch aangepast voor de gegeven architectuurinstantie, bijvoorbeeld door gegevensverbindingen tussen modellen te wijzigen, het aantal herhaalde uitvoeringen van modellen te wijzigen, of door modellen dynamisch over te slaan.

[8] SBArchOpt: https://sbarchopt.readthedocs.io/
[9] ADSG-Core: https://adsg-core.readthedocs.io/

- Architectuurinformatie wordt van ADORE naar het centrale gegevensschema[10] van het MDAO-proces overgedragen door middel van een op regels gebaseerd vertaalmechanisme.
- ADORE wordt geïntegreerd in de rekenomgeving waar het collaboratieve MDAO-proces wordt uitgevoerd door toepassing van een aangepast vraag-antwoord-patroon.

Collaboratieve MDAO voor SAO wordt gedemonstreerd door het ontwerp van een meertrapsraket.

Samengevat, door het combineren van

- **efficiënte globale optimalisatiealgoritmes** die zijn aangepast voor toepassing op SAO-problemen, resulterend in **92% minder functie-evaluaties** vergeleken met bestaande algoritmes,
- een **op functies gebaseerde methodologie voor het modelleren** van architectuur-ontwerpruimtes, om ingenieurs te ondersteunen bij het formuleren van SAO-problemen **zonder dat ze expertise in numerieke optimalisatie nodig hebben**, en
- **collaborative MDAO** uitgebreid voor het evalueren van architectuurinstanties, waardoor SAO kan worden toegepast in **grote en verspreide systeembouwprojecten**,

kunnen ingenieurs SAO toepassen om automatisch grote architectuurontwerpruimtes te verkennen om de beste architectuurinstantie voor hun ontwerpprobleem te vinden.

[10]Een centraal gegevensschema definieert hoe gegevens worden uitgewisseld tussen disciplinaire analysemodellen in een MDAO-probleem.

ZUSAMMENFASSUNG

Die Entwicklung komplexer Systeme, die zunehmend Einschränkungen in Bezug auf Nachhaltigkeit, Wirtschaftlichkeit und Sicherheit unterliegen, erfordert die Integration innovativer Technologien, die Berücksichtigung des gesamten Systemlebenszyklus und den Ausgleich widersprüchlicher Ziele der Beteiligten. Die **Systemarchitektur** beschreibt, welche *Funktionen* das System erfüllt, um die Entwurfsziele zu erreichen, und wie die *Komponenten* des Systems zusammenarbeiten, um diese Funktionen zu erfüllen. Architektonische Entscheidungen werden bereits in einem frühen Stadium des Systementwurfs getroffen und haben großen Einfluss darauf, inwieweit die Entwurfsziele durch das System erreicht werden.

Aufgrund der kombinatorischen Natur architektonischer Entscheidungen, kann der Architekturentwurfsraum, die Sammlung aller möglichen Architekturen für ein bestimmtes Entwurfsproblem, extrem groß sein. Darüber hinaus erfordert die Integration innovativer Technologien die Anwendung multidisziplinärer simulationsbasierter Bewertungen, da es an historischen Daten mangelt, die als Grundlage für Entscheidungen dienen können. Diese beiden Herausforderungen führen dazu, dass eine Entwurfsmethodik benötigt wird, die es ermöglicht, den Architekturentwurfsraum zu erkunden, ohne *alle* möglichen Architekturen bewerten zu müssen.

Die Systemarchitekturoptimierung (SAO) tut genau das: Sie kombiniert numerische Optimierungsalgorithmen zur automatischen Durchsuchung des Architekturentwurfsraums (“Architekturgenerator”) mit einer multidisziplinären, simulationsbasierten Bewertung von Architekturkandidaten (“Architekturbewerter”).

Der **Architekturgenerator** besteht aus zwei Elementen:

- Eine Möglichkeit für den Systemarchitekten, **den Architekturentwurfsraum zu modellieren** und das Optimierungsproblem zu formulieren.

  Das Modell des Architekturentwurfsraums sollte in der Lage sein, Entscheidungen für die Funktionszerlegung, die Zuordnung von Funktionen zu Komponenten, die Charakterisierung und Spezialisierung von Funktionen sowie Komponenten, und die Verbindungen zwischen Komponenten darzustellen. Sobald das Optimierungsproblem der Architektur definiert ist, sollte es automatisch in ein Optimierungsproblem codiert und die generierten Entwurfsvektoren in Architekturinstanzen decodiert werden können.

  Derzeit gibt es keine Kombination aus Modellierungssprache und Kodierungs- und Dekodierungsalgorithmen, die alle oben genannten Herausforderungen adressiert.

- **Optimierungsalgorithmen**, die das Optimierungsproblem lösen können.

  SAO-Probleme sind anspruchsvoll, da sie im Allgemeinen gekennzeichnet sind durch gemischt-diskrete Entwurfsvariablen, eine Hierarchie von Entwurfsvariablen, Blackbox- sowie aufwendige Bewertungsfunktionen, mehrere widersprüch-

liche Zielfunktionen, Entwurfsnebenbedingungen und verborgene Nebenbedingungen. Die Hierarchie der Entwurfsvariablen ergibt sich aus Aktivierungsbeziehungen (Variablen, die bestimmen, ob andere Variablen aktiv sind) und aus Wertbedingungen (Variablen, welche die verfügbaren Optionen anderer Variablen begrenzen). Evolutionäre Algorithmen und Bayes'sche Optimierungsalgorithmen sind globale Optimierungsalgorithmen, die solche Probleme lösen können.

Bestehende Abtastalgorithmen[11] *berücksichtigen jedoch nicht explizit die hierarchische Natur des Entwurfsraums, sodass es zu einer Über- oder Unterabtastung bestimmter Regionen im Entwurfsraum kommen kann. Außerdem ist nicht klar, wie sich der Korrekturalgorithmus*[12] *auf die Optimierungsleistung auswirkt. Schließlich wurde die Bayes'sche Optimierung noch nicht für alle Herausforderungen von SAO-Problemen gleichzeitig demonstriert.*

Der **Architekturbewerter** berechnet die Leistung einer gegebenen Architekturinstanz. Der multidisziplinäre Charakter des Systementwurfs und die Notwendigkeit einer simulationsbasierten Bewertung erfordern die Anwendung der Multidisziplinären Entwurfsanalyse und -Optimierung (MDAO). Die kollaborative MDAO erweitert diesen Ansatz, indem sie es ermöglicht, verschiedene Analysemodelle zu verknüpfen, die von verteilten Teams und/oder Organisationen in großen Systementwurfsprojekten entwickelt, verwaltet und ausgeführt werden.

Derzeit sind die kollaborativen MDAO-Prozesse statisch. Ihr Verhalten sollte jedoch flexibel sein, um alle generierten Architekturinstanzen zu berücksichtigen. Außerdem sollten alle relevanten Architekturinformationen an den MDAO-Prozess weitergegeben werden und nicht nur numerische Parameterwerte, wie es heute der Fall ist. Schließlich sollte der Architekturgenerator in dieselbe Rechenumgebung integriert werden, in der der MDAO-Prozess ausgeführt wird.

Diese Arbeit trägt zur praktischen Anwendung von SAO bei, indem sie die oben genannten wissenschaftlichen Lücken schließt.

OPTIMIERUNGSALGORITHMEN FÜR SAO

Im ersten Teil wurden effiziente globale **Optimierungsalgorithmen für SAO** über folgende Schritte anwendbar gemacht:

- die Entwicklung eines hierarchischen Abtastalgorithmus, der ein Über- oder eine Unterabtastung einiger Regionen im Entwurfsraum vermeidet;
- zeigen, dass eine problemspezifische Korrektur genügt: Problemagnostische Korrekturalgorithmen verbessern die Optimierungsleistung nicht wesentlich;
- die Integration von Informationen über den hierarchischen Entwurfsraum in die Optimierungsalgorithmen, um die Generierung gültiger Entwurfsvektoren zu unterstützen und Informationen über die Aktivität für die Abtastung und die Erstellung von Ersatzmodellen (für die Bayes'sche Optimierung) zu nutzen; und

[11] Ein Abtastalgorithmus generiert den Anfangssatz von Architekturen, um globale Optimierungsalgorithmen zu initialisieren.
[12] Ein Korrekturalgorithmus stellt sicher, dass die Wertbedingungen eingehalten werden.

- die Entwicklung einer Strategie für den Umgang mit verborgenen Nebenbedingungen (aufgrund von Simulationsmiserfolgen) für die Bayes'sche Optimierung durch die Vorhersage der Wahrscheinlichkeit, dass ausgewählte Einfügepunkte simulierbar sind.

Die Entwicklungen wurden anhand von drei Testproblemen und der Optimierung einer Triebwerksarchitektur mit verborgenen Nebenbedingungen demonstriert. Es hat sich gezeigt, dass der Bayes'sche Optimierungsalgorithmus SAO-Probleme mit bis zu 92% weniger Funktionsauswertungen im Vergleich zu evolutionären Algorithmen lösen kann. In der Praxis bedeutet dies, dass Optimierungsergebnisse früher zur Verfügung stehen oder dass ein größerer Entwurfsraum in derselben Zeit erkundet werden kann. Die Optimierungsalgorithmen und Testprobleme sind open-source verfügbar[13].

MODELLIERUNG VON SAO-PROBLEMEN

Im zweiten Teil wurde eine Methodik zur **Modellierung von SAO-Problemen** entwickelt, die aus folgenden Elementen besteht:

- der Entwurfsraumgraph für die Modellierung hierarchischer Auswahl- und Verbindungsentscheidungen, für die automatische Kodierung dieser Entscheidungen als Entwurfsvariablen und für die automatische Dekodierung von Entwurfsvektoren zu Graphinstanzen;
- der Architekturentwurfsraumgraph, welcher den Entwurfsraumgraph um Knotentypen und Regeln erweitert, die für die Systemarchitektur spezifisch sind, wie z.B. Funktionen, Komponenten und Ports; und
- ADORE, eine webbasierte grafische Benutzeroberfläche für die Erstellung und Untersuchung des Architekturentwurfsraumgraphens, sowie verschiedene Schnittstellen zur Anbindung von Bewertungscode und Optimierungsalgorithmen.

Der Entwurfsraumgraph ist open-source verfügbar[14].

Es wurde ein funktionsbasierter Bottom-up-Prozess definiert, der Ingenieure bei der Anwendung von ADORE zur Modellierung von SAO-Problemen unterstützt. Es wurde gezeigt, dass dieser Prozess im Vergleich zu bestehenden Top-down-Prozessen zu einem natürlicheren Ansatz führt. Die Methodik wurde anhand eines SAO-Problems für hybrid-elektrische Antriebe demonstriert. Eine zusätzliche Studie mit drei Testproblemen zeigt, dass von ADORE definierte und kodierte Optimierungsprobleme genauso gut oder besser abschneiden als manuell formulierte Optimierungsprobleme.

KOLLABORATIVE MDAO FÜR SAO

Im dritten Teil wurde die **kollaborative MDAO** für den Einsatz bei SAO-Problemen erweitert:

- Das Verhalten des kollaborativen MDAO-Prozesses wird während der Laufzeit für die gegebene Architekturinstanz dynamisch angepasst, z.B. durch die Änderung der Datenverbindungen zwischen Modellen, die Änderung der Anzahl

[13]SBArchOpt: https://sbarchopt.readthedocs.io/
[14]ADSG-Core: https://adsg-core.readthedocs.io/

sich wiederholender Ausführungen von Modellen oder durch das dynamische Überspringen von Modellen.

- Die Architekturinformationen werden mit Hilfe eines regelbasierten Übersetzungsmechanismus von ADORE in das zentrale Datenschema[15] des MDAO-Prozesses übertragen.
- ADORE wird in die Rechenumgebung integriert, in welcher der kollaborative MDAO-Prozess ausgeführt wird, indem ein modifiziertes Frage-Antwort-Muster angewendet wird.

Die kollaborative MDAO für SAO wird durch den Entwurf einer mehrstufigen Rakete demonstriert.

Zusammengefasst lässt sich sagen, dass Ingenieure durch die Kombination von

- **effiziente globale Optimierungsalgorithmen**, die für die Anwendung auf SAO-Probleme angepasst sind, was zu **92% weniger Funktionsauswertungen** im Vergleich zu bestehenden Algorithmen führt,
- eine **funktionsbasierte Methodik zur Modellierung** von Architekturentwurfsräumen, zur Unterstützung von Ingenieuren bei der Formulierung von SAO-Problemen, **ohne dass Fachkenntnisse in numerischer Optimierung erforderlich sind**, und
- **kollaborative MDAO** erweitert für die Bewertung von Architekturinstanzen, so dass SAO in **großen und verteilten Systementwurfsprojekten** angewendet werden kann,

SAO anwenden können, um automatisch große Architekturentwurfsräume zu erkunden und die beste Architekturinstanz für ihr Entwurfsproblem zu finden.

[15] Ein zentrales Datenschema definiert, wie Daten zwischen disziplinären Analysemodellen in einem MDAO-Problem ausgetauscht werden.

Nomenclature

Acronyms

ADORE	Architecture Design and Optimization Reasoning Environment
ADSG	Architecture Design Space Graph
API	Application Programming Interface
BO	Bayesian Optimization
CCF	Connection Choice Formulation
CDS	Central Data Schema
CFE	Class Factory Evaluator
CR	Correction Ratio
CRF	Correction Fraction
CT	Correction Time
DoE	Design of Experiments
DSG	Design Space Graph
EGO	Efficient Global Optimization
FR	Failure Rate
G2L	Global-to-Local
GNC	Guidance, Navigation & Control
GP	Gaussian Process
GUI	Graphical User Interface
HEP	Hybrid-Electric Propulsion
HV	Hypervolume
IR	Imputation Ratio
KBE	Knowledge-Based Engineering
LHS	Latin Hypercube Sampling
MBSE	Model-Based Systems Engineering
MD	Mixed-Discrete
MDAO	Multidisciplinary Design Analysis and Optimization
MDAx	MDAO Workflow Design Accelerator
MOEA	Multi-Objective Evolutionary Algorithm
MRD	Maximum Rate Diversity
NFE	Node Factory Evaluator

NSGA-II	Non-dominated Sorting Genetic Algorithm II
OPM	Object Process Methodology
PIDO	Process Integration and Design Optimization
PLE	Product Line Engineering
PoV	Probability of Viability
QOI	Quantity of Interest
RBF	Radial Basis Function
RCE	Remote Component Environment
RD	Rate Diversity
RFC	Random Forest Classifier
SAO	System Architecture Optimization
SBD	Set-Based Design
SBO	Surrogate-Based Optimization
SE	Systems Engineering
SMT	Surrogate Modeling Toolbox
SysML	Systems Modeling Language
TSFC	Thrust-Specific Fuel Consumption
XDSM	Extended Design Structure Matrix

SYMBOLS

$*_{\text{valid,discr}}$	Set of valid discrete design vectors $\boldsymbol{x}$ or activeness information δ
$\boldsymbol{x}$	Design vector
δ	Activeness function
Dcorr	Distance correlation
f	Objective function
G	Grouping matrix
g	Design constraint (inequality)
M	Connection matrix
$M_{\text{all,valid}}$	All valid connection matrices
n_f	Number of objectives
n_g	Number of inequality constraints
N_j	Number of discrete design variable options
N_{fe}	Number of function evaluations
n_{x_c}	Number of continuous design variables
n_{x_d}	Number of discrete design variables
x	Design variable

1 INTRODUCTION

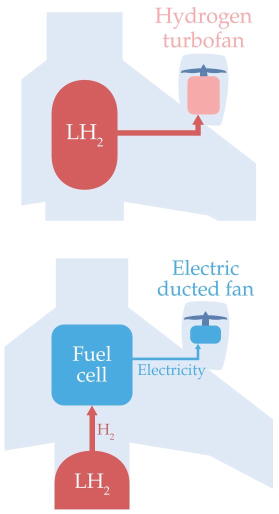
CHAPTER CONTENTS

SYSTEMS developed in the future will be subject to more stringent requirements, such as sustainability [31], economic [32], and safety [33] requirements, and will operate in environments characterized by high uncertainty [34]. To meet these requirements, there is a need to integrate disruptive technologies [35, 36], while considering the entire system life-cycle [37, 38] and weighting conflicting stakeholder interests [39]. Take for example the development of aircraft propulsion systems:

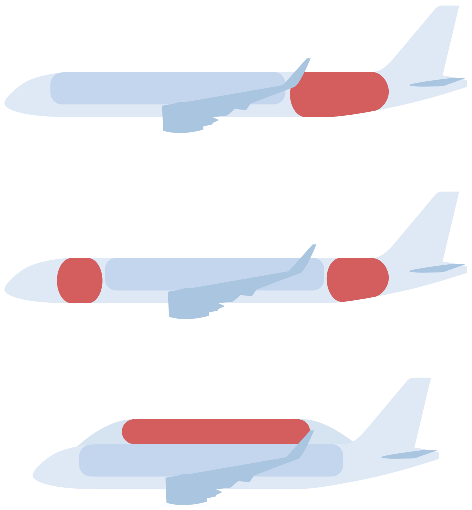
- Thrust can be generated by propellers, turbofans, open rotors, or combinations of these [37, 40].
- Propulsion units can be integrated at different locations on the aircraft, and can be installed in different amounts (e.g. distributed electric propulsion architectures [40]).
- Mechanical power can be generated by turbines (either burning conventional fuels or novel fuels such as hydrogen), by electric motors (supplied by batteries, turbines, fuel cells, or a combination), or hybrid-electric architectures [33, 40].

This chapter is based on [1, 4–6, 8].

- Energy can be stored in conventional fuels, in sustainable fuels such as hydrogen (see Figure 1.1), in batteries, or again a combination [33, 40, 41].
- Connections from energy storage, to energy distribution, to energy consumption components can be established in various ways, all representing different trade-offs (e.g. reliability vs. weight [42]).
- Conventional fuels, hydrogen, and electricity can be sustainably produced from different sources [43, 44], and using various chemical processes [41, 45].



(a) Direct hydrogen combustion (top) and fuel cell (bottom) propulsion architectures.



(b) Comparison of various hydrogen tank (shown in red) locations in the aircraft.

Figure 1.1: Hydrogen propulsion and tank integration architectures. Figure adapted from [33].

The selection of components or technologies to include in a system, and how these collaborate to achieve the system goals is described by the *system architecture* [39]. The number of components, their specializations, and connections between components are also part of the system architecture description [39].

The set of all possible architectures for a given design problem is referred to as the *architecture design space*, which is combinatorial in nature and can be extremely large, making an exhaustive assessment and comparison of all possible architectures infeasible [46, 47]. Additionally, due to the push for more innovative technologies for which historical data are lacking, there is a need to apply multidisciplinary, simulation-based evaluation earlier in the design phase [48]. Architectural choices need to be considered holistically, because in general, combinations of choices affect the system and its (conflicting) performance metrics in non-trivial and non-linear ways [47, 49]. These challenges drive the need for a methodology that allows considering different architectures in the early design phase without the need to consider *all* architectures, as that would be infeasible.

This research work explores the application of *System Architecture Optimization* (SAO) to address these challenges. SAO formulates the architecting process as a numerical optimization problem, where an optimization algorithm automatically searches the design space to find the optimal architecture(s) for the design problem at hand. Applying SAO therefore requires:

- the architect to be able to specify the SAO problem;
- the application of multidisciplinary, simulation-based evaluation; and
- optimization algorithms that can solve SAO problems.

The following sections start by introducing the context in which this research was performed in Section 1.1. Then, Section 1.2 places SAO in the wider systems engineering context to show where the system architecture plays a role. System architecture and how it is defined is treated in more details in Section 1.3, and Section 1.4 discusses the need for quantitative performance evaluation in the design process. From there, Section 1.5 introduces SAO in more details, and identifies the science gaps addressed by this work. The research questions and objectives are finally presented in Section 1.6.

1.1. RESEARCH CONTEXT

The development of the SAO methodologies and tools presented in this work was partially done in two EU-funded international research projects: AGILE 4.0 (Towards Cyber-Physical Collaborative Aircraft Development) and COLOSSUS (Collaborative System of Systems Exploration of Aviation Products, Services & Business Models).

AGILE 4.0[1] [16] ran from 2019 to 2023, with a high-level objective to investigate new methodologies and tools to accelerate the development process of complex aviation systems, including their supporting production and certification systems. To do so, existing collaborative Multidisciplinary Design Analysis and Optimization (MDAO) capabilities were integrated with Model-Based Systems Engineering (MBSE). The system architecture forms the bridge between the two, and was envisioned to be a driver of the optimization process by enabling the automated generation and evaluation of system architectures. This technical goal is what lead to the developments towards System Architecture Optimization (SAO) as presented in this dissertation. In the AGILE 4.0 project, the first versions of the methodologies and tools were tested in seven application cases (ranging from hybrid-electric propulsion to the design of manufacturing systems), with varying levels of integration of SAO into the MDAO process.

The objective of COLOSSUS[2] [50] (running from 2023 to 2026) is to develop System-of-Systems (SoS) evaluation methodologies and tools, applied to sustainable urban mobility and wildfire fighting application cases. SoS are a form of complex systems characterized by the mutual interplay of several independently operated and evolving systems, thereby increasing development and operational complexity. In the COLOSSUS project, SAO is applied and further developed to support modeling and exploring SoS architecture design spaces. Also, optimization algorithms for solving SAO problems are further developed.

[1] https://www.agile4.eu/
[2] https://colossus-sos-project.eu/

1.2. (MODEL-BASED) SYSTEMS ENGINEERING

INCOSE and ISO 15288 define *Systems Engineering* (SE) as follows [34, 51]:

> Systems Engineering is a transdisciplinary and integrative approach to enable the successful realization, use, and retirement of engineered systems, using systems principles and concepts, and scientific, technological, and management methods.

with a *system* being a combination of elements that together provide functions (or capabilities) that the individual elements do not [34, 38]. SE considers the system holistically, ensuring that contributions of individual engineering disciplines are balanced, to produce a consistent result not dominated by one domain over another [38]. The overall goal of SE is to realize systems that accomplish their intended purposes, are resilient to operational uncertainty, and minimize unintended consequences [34].

The SE process proceeds from the whole to the details, and structures the development and implementation process according to various life-cycle phases [34, 47, 51]. First, the purpose of the system is determined by eliciting stakeholder needs. Needs are formalized into system requirements, which form the basis for the definition of the system architecture. The design process then proceeds with lower-level, more detailed design phases, which are at later stages integrated together to realize the higher-level system design. Detailed design and implementation also feature extensive verification and validation against system requirements and stakeholder needs, to ensure the system will fulfill its purpose. This process is commonly represented in the Vee-model [34], shown in Figure 1.2.

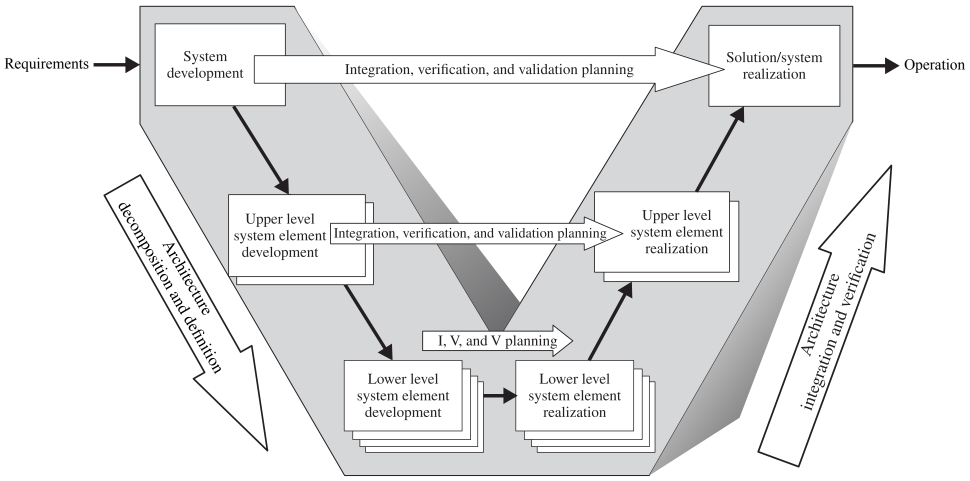

Requirements
System development
Integration, verification, and validation planning
Solution/system realization
Operation
Upper level system element development
Integration, verification, and validation planning
Upper level system element realization
I, V, and V planning
Lower level system element development
Lower level system element realization
Architecture decomposition and definition
Architecture integration and verification

Figure 1.2: The systems engineering Vee-model, showing the architecture definition process on the left side. Figure adapted from [34].

Traditionally, the systems engineering process has been implemented by capturing information in non-semantic and unlinked documents: the document-based systems engineering approach [34]. In contrast, *Model-Based Systems Engineering* (MBSE) aims to improve the development process of complex systems by leveraging models in all life-cycle phases, to enhance the ability to represent, analyze, share and manage information about the system [34, 52]. MBSE revolves around the creation and management of a descriptive system model, which includes data about the system context, requirements, architecture, behavior, parameters, and verification activities. The model is machine readable, which opens up the potential for increasing productivity by automation [39, 52, 53], for example by automatically generating documentation from models, automated consistency checks, automated verification and validation, and automation in the design process itself. If applied well, MBSE results in improved communication among stakeholders, increased ability to manage complex systems, improved product quality, reduced development time, reduced risk, and improved knowledge reuse [52].

The system model integrates information from all involved engineering disciplines, which are often captured in other types of models, such as geometric and analysis models, to describe other aspects of the system and analyze its performance. Care is taken that all integrated information is consistent: i.e. based on shared assumptions and common terminology. Systems engineers interact with models through views [52], which only show relevant information for a subset of stakeholders, and limit the amount of information displayed to manage human cognitive limitations [39]. The types of artifacts and relationships between artifacts that can be represented in models are specified by MBSE languages. The main standardized MBSE languages currently are SysML (Systems Modeling Language) [54–56], OPM (Object Process Methodology) [57, 58], and Arcadia (Architecture Analysis and Design Integrated Approach) [59, 60].

1.3. Defining the System Architecture

System architecture is the description of what components a system consists of, what functions they perform (i.e. why they are there), and how they are connected and related to each other [39]. It provides a bridge between upstream systems engineering activities, like identifying stakeholders, needs, and requirements, and downstream phases like detailed design and operation of the system (see Figure 1.2). The *architect* is responsible for defining and managing the system architecture, and does so by [39]:

- *Reducing ambiguity* from upstream activities, by defining the boundary, goals, and functions of the system through interactions with system stakeholders (e.g. customers, strategy departments, authorities).
- *Defining the system architecture* (i.e. the "system architecting" activity), by defining key performance metrics, developing architecture options, conducting trade-studies and optimization, and selecting an architecture to continue with in downstream design phases.
- *Managing complexity* to ensure the system can be understood by all stakeholders, by decomposing form and function, defining interfaces between (sub)systems and the system context, configuring subsystems, defining the degree of modularity, balancing flexibility and optimality, and balancing make-or-buy decisions.

As mentioned in the last point, the system should be decomposed in some way in order to manage complexity when defining the system architecture. Compared to domain-specific system decomposition such as by engineering discipline (e.g. aerodynamics, structures) or by ATA (Air Transport Association) chapters [61], functions provide the basis for decomposition in system architecting [62, 63]: a function-based decomposition provides a generic breakdown of functions to be satisfied by the system. Functions are independent of the system architecture, specific solutions or technology. Additionally, function-based decomposition enables explicit traceability from functional requirements to the system architecture.

With a function-based decomposition technique, care is taken to define *what* the system does (its functions) before *how* the system looks like (its form) [63]. Form is the artifact that will be implemented in the physical system to fulfill the functions. Elements of form are also referred to as system components. Functions describe what the system should be able to do in order to meet stakeholder needs and functional requirements, and ultimately provide the reason the system exists. Functions represent a process (verb) applied to an operand (e.g. material, energy, or signal flow) [39]. For cyber-physical systems, HIRTZ ET AL. [64] provide a basis for solution-neutral processes and operands. Examples of functions are "accelerate air", "process signal" and "generate power".

The system architecture defines how functions are fulfilled, by assigning components to functions, based on available knowledge about which components can fulfill which functions. Additionally, it is possible that components themselves need additional functions to be performed within the system, thereby requiring the inclusion of additional components. This iterative "zig-zagging" procedure is completed once all functions have been fulfilled. The finalized system architecture describes this function-form allocation, together with relationships among the components [39]: it is a function-form-structure mapping. This principle is visualized in Figure 1.3.

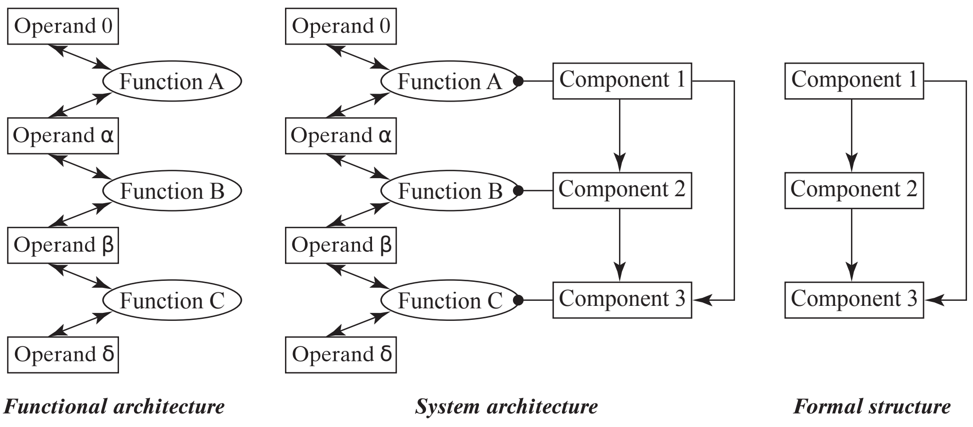


Figure 1.3: System architecture as the combination of functional architecture and the elements and structure of form (i.e. the components). Figure adapted from [39].

By separating the architecting activity in solution alternatives generation and performance evaluation phases, bias towards selecting conventional or familiar architectures is reduced [39, 65]. The observation that deferring decision-making until after many solution alternatives have been systematically generated leads to better designs in the end, has long been known in the fields of design thinking [66] and Set-Based Design (SBD) [67]. It is also observed that keeping the design space open further into the design process, results in designs that are more robust to changes in requirements or environmental factors (e.g. market, regulations) [68].

The space encompassing all possible architectures that can be generated for a given problem is known as the *architecture design space*, and is defined by *architectural choices and constraints*. According to CRAWLEY ET AL. [39], types of architectural choices include (illustrated by the propulsion system example given earlier in this chapter):

- *Decomposing* form and function. For example, a decomposition of aircraft-level to propulsion-system-level functions, or a decomposition of a propulsion unit into its components (propeller, gearbox, power converter, etc.).
- *Mapping* function to form. An example of this is the allocation of the function to generate thrust to either propellers, turbofans, open rotors, or a combination of these.
- *Specializing and characterizing* form and function. For example the selection of the number of instances of some component to include (e.g. the number of engines on a wing), or the selection of some specific off-the-shelf electric motor.
- *Connecting* form. For example, choosing the different connection pathways from energy storage, to energy distribution, to energy consumption components.

Architectural constraints restrict which selection of choices are compatible with each other, often defined in the form of "requires" constraints (one element requiring the inclusion of some other element) or "exclusion" or "incompatibility" constraints [61, 69, 70]. Architectures can then be generated by combining architectural choice options, while ensuring architectural constraints are satisfied.

Architectures may further be parameterized by *architecture-specific design parameters* [49]. These parameters do not modify the architecture itself, however do influence performance or even feasibility of the architecture. Therefore, they are relevant for the architecting process, and are also part of the architecture design space. Examples of architecture-specific design parameters are sizing parameters such as rated power or bypass ratio, or operational parameters such as hybridization ratio or cruise speed. Together, architectural choices and architecture-specific design parameters are called the *architecture design choices*. An architecture with values assigned to its design parameters is called an *architecture instance*, as it represents a specific instantiation of the architecture. If an architecture contains no design parameters, the architecture only contains one architecture instance. Figure 1.4 visualizes how the architecture design space, architectures, and architecture instances are related.

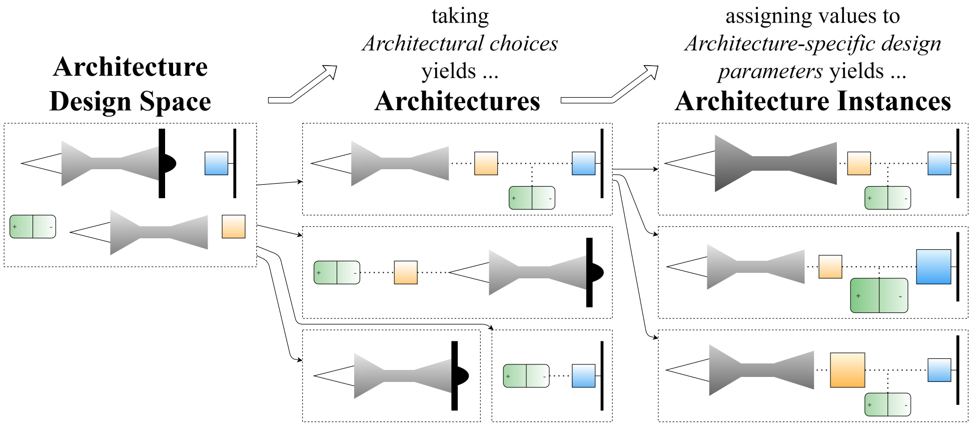

Architecture Design Space
taking *Architectural choices* yields ...
Architectures
assigning values to *Architecture-specific design parameters* yields ...
Architecture Instances

Figure 1.4: Starting from the architectural design space, taking architectural choices yields the architectures. Starting from an architecture, assigning values to architecture-specific design parameters yields the architecture instances.

Observations:

- *Defining system architectures based on functions promotes neutrality when defining and selecting solutions, and enables traceability to functional requirements.*
- *Taking a structured approach to architecture instance creation by first explicitly defining architectural choices before generating and evaluating solution alternatives can improve the design process, resulting in less biased and more robust final designs.*

1.4. AUTOMATED QUANTITATIVE PERFORMANCE EVALUATION

Decisions made when defining the system architecture have a significant impact on the performance of the final system, i.e. how well the system meets its requirements. However, during the design stage, limited knowledge about the system behavior is available, making it challenging to make informed decisions. As the design process progresses, more becomes known about the behavior of the system and the impact of decisions, however it also becomes more difficult to retroactively change decisions taken earlier in the design process. This phenomenon has been called the *knowledge paradox* [71, 72], which is part of the reason that SBD places much focus on keeping the design space open further into the design process [67].

Additionally, and especially for complex systems, the architecture design space, can be extremely large. This is because during the architecting phase, choices are mostly of a discrete nature: for example, a choice between technology alternatives. Already for a low number of discrete choices, a design space can suffer from a *combinatorial explosion of alternatives* [46, 47]. For such design spaces, it is infeasible to exhaustively search all possible combinations of options.

Involving expert judgement and experience can support taking decisions in such uncertain and large design spaces, however, may suffer from expert bias, subjectivity, conservatism or overconfidence [39, 47, 73, 74], and cannot be used for novel systems that bear little resemblance to existing systems [48]. Instead, performance metrics and goals should be formulated first [75], which should then be used to compare generated alternatives in the search for the "best" solution(s) to the problem [47]. The identification of the best solution(s) can only be done for such large design spaces if the comparison can be done without designer interaction, which requires that the performance metrics [76]:

- are *quantitative* in nature, such that "better" can be expressed as the minimization or maximization of some performance metric; and
- can be calculated *automatically*, to enable exploring a large design space within an appropriate time.

When designing complex systems, the views and considerations of diverse engineering disciplines should be considered [38]. Often, these disciplines are strongly connected, meaning that it is not possible to design the system by considering each discipline in sequence, but rather all disciplines should be considered simultaneously to obtain a feasible, let alone optimal, design [72]. Therefore, the performance metrics should be evaluated keeping the multidisciplinary nature of systems engineering in mind.

Multidisciplinary Design Analysis & Optimization (MDAO) provides techniques and tools for doing exactly that [72]: it enables coupling disciplinary analysis tools in automated calculation workflows, thereby ensuring that all contributions of individual engineering disciplines are balanced and consistent with each other while analyzing and optimizing system-level performance. Figure 1.5 shows an example of a coupled MDAO system. MDAO has been identified as an important technology to support MBSE [53, 77, 78], and thus by extension also for SAO.

Observations:

- *Quantitatively evaluating system architecture performance mitigates the knowledge paradox and reduces bias.*
- *Automating the calculation of performance metrics enables searching a large design space.*
- *Considering the multidisciplinary nature of systems engineering when calculating performance metrics ensures that a realistic, balanced design is obtained.*

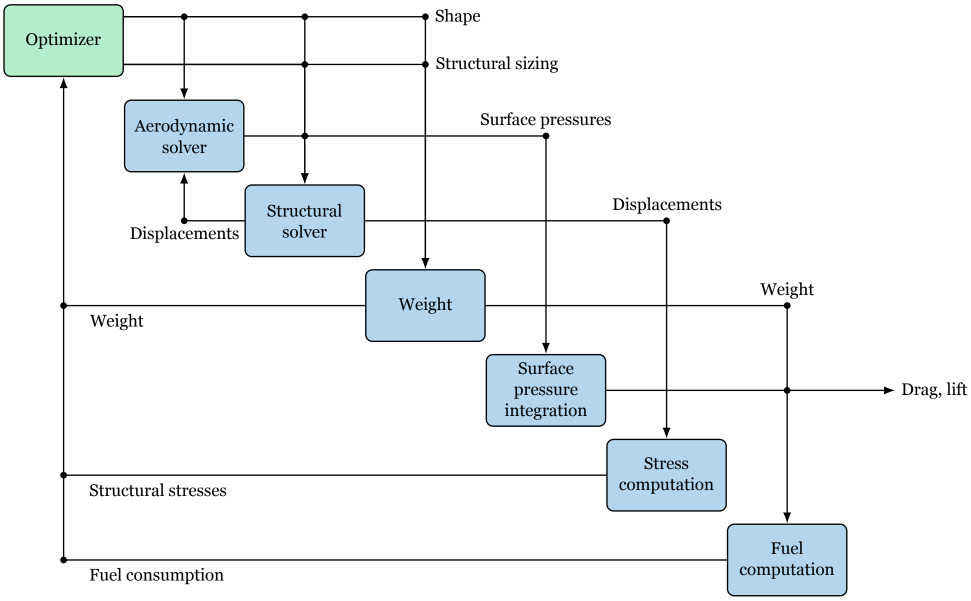


Figure 1.5: Example of a coupled computational system (as is typical for Multidisciplinary Design Analysis and Optimization (MDAO)), driven by an optimizer. Figure adapted from [49].

1.5. System Architecture Optimization

System Architecture Optimization (SAO) combines a structured approach for creating architecture instances (Section 1.3) with automated quantitative performance evaluation (Section 1.4) to search architecture design spaces. It does so by formulating the architecting process as a numerical optimization problem, and applying optimization algorithms to suggest architecture instances to evaluate [79]. Conceptually, any optimization problem consists of two elements: an algorithm to suggest design solutions (the optimization algorithm) and a function to tell the algorithm how good a given solution is (the evaluation function) [49]. Similarly, a system architecture optimization problem consists of [80]:

1. a mechanism to suggest architecture instances to be evaluated, the *architecture generator*; and
2. a function to evaluate the performance of a given architecture instance, the *architecture evaluator*.

In SAO, the architecture generator is driven by an optimization algorithm, which requires a formal definition of the architecture design space. The following sections discuss modeling the architecture design space (Section 1.5.1), how to use that to generate architecture instances (Section 1.5.2). Then, salient characteristics of SAO problems (Section 1.5.3) and appropriate classes of optimization algorithms (Section 1.5.4) are discussed. Section 1.5.5 discusses the architecture evaluator in more details.

1.5.1. ARCHITECTURE DESIGN SPACE MODELING

The architecture generator is an automated version of the architecture synthesis [47] or concept generation [39, 81] step in systems engineering. It generates architecture instances from an associated *architecture design space*. As introduced in Section 1.3, the architecture design space is defined by architectural choices and architecture-specific design parameters (when combined, known as architecture design parameters), and constrained by architectural constraints. The design space should be specified formally enough so that automatic reasoning by a computer program is possible. However, it should be possible to do so without requiring the systems engineer to have expertise in numerical optimization. This can be achieved by using function, form, and other systems engineering concepts, instead of optimization jargon, to define the design space. This additionally enables the smooth integration of SAO in the MBSE process and maintain traceability from requirements to elements in architecture instances [63].

A system model created in an MBSE setting typically represents a specific architecture instance or several architecture instances [63, 65]. Although useful for a variety of reasons [82], such models by themselves do not represent architecture design spaces and therefore cannot be used as a basis for architecture optimization. To model an architecture design space, architecture design choices should be modeled explicitly. In literature, choice modeling is also known as *variability modeling*, a term originating from the (software) Product Line Engineering (PLE) domain [69, 83].

Modeling choices in an MBSE context is made possible by specific extensions of the Systems Modeling Language (SysML), such as Common Variability Language (CVL) [84, 85], Variant-Specific Stereotypes (VSS) [86], Variant Modeling with SysML (VAMOS) [87], or other custom approaches [65, 76, 88]. Variability is an integral part of SysMLv2 [89–91], see Figure 1.6 for an example, showing that it is considered an important capability to be supported in the future.

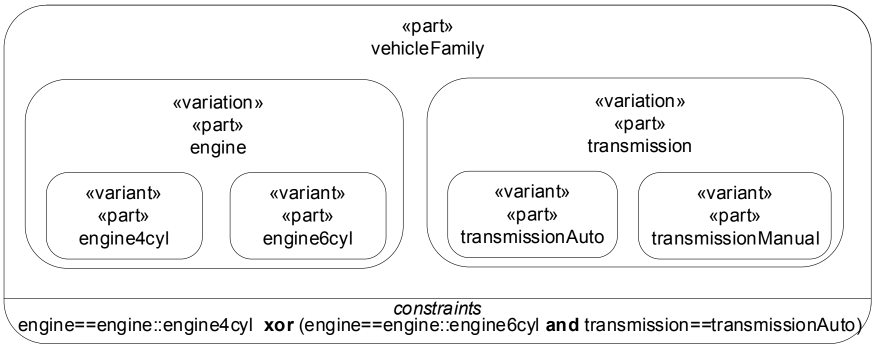


Figure 1.6: Variability model of a vehicle family with different variants for the engine and transmission, modeled using SysMLv2. Figure reproduced from [92].

Another approach is by creating a dedicated variability model that specifies how a 150% architecture model (a model containing all architecture elements in the design space) is transformed to an architecture instance [93]. This can for example be done using feature models [69, 94, 95], of which Figure 1.7 shows an example. Feature models are trees of choices, where each choice represents some "feature" of the object under design, and choices select which features are implemented by selecting from a list of options (often mutually-exclusive, however also other cardinalities are possible [94]). Each selected option can then activate further feature selection choices, and/or pose constraints on other choices such as requiring or preventing some option to be chosen. Feature models were originally developed for modeling software variability [83], however have also found application in MBSE for the design of (cyber-)physical systems, known as Model-Based PLE (MBPLE) [96–98]. Notably, Airbus is developing and deploying MBPLE methods for modeling variability in system architectures as part of their Digital Design Manufacturing and Services (DDMS) program [99].

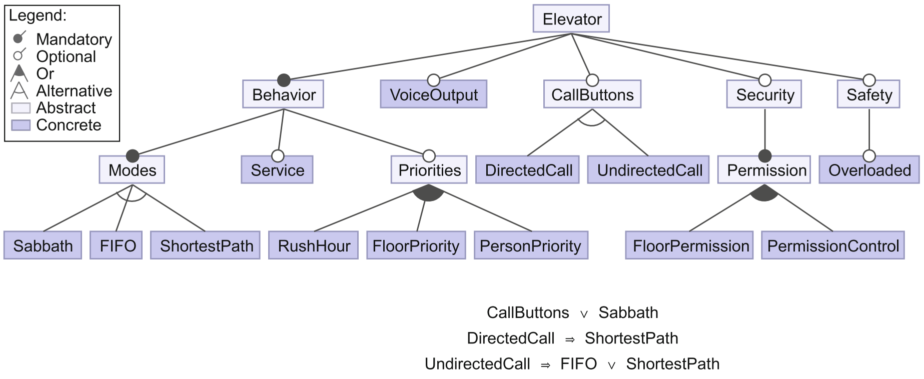


Figure 1.7: Feature model of an elevator system, showing different options for different features. Cross-tree constraints are shown near the bottom. Figure reproduced from [95].

Function-means models [70, 100] and configurable components [101] offer a hybrid between dedicated systems modeling languages and variability models, incorporating variability and systems elements (function and form) in one model.

Observations:

Specifying the system architecture design space based on functions and other systems engineering terminology enables integration in the MBSE process and maintain traceability from requirements to elements in architecture instances.

1.5.2. GENERATING ARCHITECTURE INSTANCES

Generating architecture instances from some architecture design space model can either be done exhaustively (architecture enumeration) or selectively (architecture optimization). Several *architecture enumeration* methods based on graph grammars [102–104] have been developed in the past, including ArchEx [105], perfect matchings [106, 107], RoMOGA [108], and Automatic Topology Generation [109]. In such approaches, graph nodes represent architecture elements and edges represent some connection between elements; graph structure constraints are formulated to ensure established connections are feasible. Siemens Simcenter Studio enables architecture enumeration by modeling architecture elements and port connection constraints, and then solving a Constraint Satisfaction Problem (CSP) [110–112]: Figure 1.8 shows an example architecture design space model.

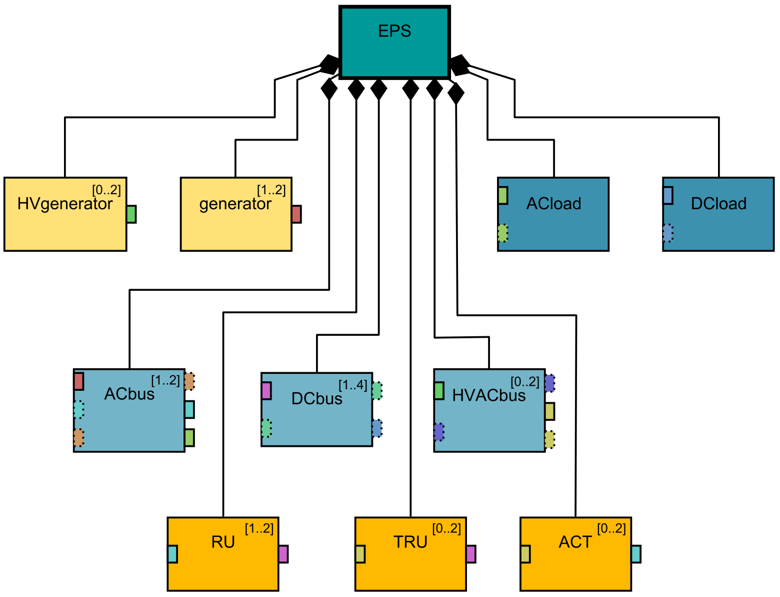


Figure 1.8: Architecture model created in Siemens Simcenter Studio that models an Electric Power System (EPS) with components that can be connected to each other by establishing connections between ports of the same color. Component cardinality (instantiation) choices are shown in square brackets. Figure reproduced from [110].

The morphological matrix is a well-developed method for modeling system variability. Table 1.1 shows an example morphological matrix: it contains 14 architecture design choices with up to 5 options each, with a total of 144 million different combinations. Usually also some way to model (in)compatibilities between options is needed before the morphological matrix can be used to enumerate architecture instances: an (in)compatibility matrix is used to do this by ARMA [61], IRMA [113], and [114]; [115] uses a Constraint Satisfaction Problem (CSP) formulation instead. Other methods for architecture enumeration include functional flows [42, 116, 117], resource flows [118], Architecture Decision Graph (ADG) [119], RAAM [46], and using description logic reasoners [120].

The main downside of architecture enumeration is that architecture design spaces can be extremely large: for example 144 million architectures for the design space shown in Table 1.1, or 79 million architectures in the Guidance, Navigation and Control (GN&C) system architecture problem in [122]. This makes architecture enumeration impractical and/or infeasible due to time or computational resource constraints [73], which can stem from the enumeration process, the architecture evaluations, or both. Especially architecture evaluation can be constraining, due to a need for simulation-based simulation for novel systems as discussed previously. A more practical approach therefore is to

Table 1.1: An example morphological matrix of an aircraft. Contents are based on the morphological matrix from [121].

Choice	Options				
Configuration	Tube and Wing	BWB			
Fuselage Cross-section	Elliptical	Circular			
Wing Shape	Elliptical	Rectangular	Diamond	Triangular	
Wing Sweep	Forward	Backward	Variable	Straight	Switch
Wing Structure	Internal	Truss	Strut	Cable	
Materials	Aluminum	Steel	Composites		
High Lift Devices	Slats	Flaps	Both		
Nr of Engines	1	2	3	4	
Engine Type	Turboprop	Turbofan	Turbojet	Propfan	Piston
Engine Position	Over Wing	Under Wing	Fuselage	Tail	Embedded
Fuel Type	Conventional	Biofuels	Synthetic	Hydrogen	Fuel Cells
Takeoff	Traditional	Floating	Assisted	Vertical	
Landing	Traditional	Assisted	Vertical	Spiral	Steep
Piloting	Manned	Auto	Ground Pilots		

selectively generate architectures and only evaluate these to incrementally explore the design space. *Architecture optimization* indeed selectively generates architectures and uses evaluation results to steer the exploration towards the best architecture(s).

Feature models [123–125], variability modeling in SysML [85, 88], and the morphological matrix [79, 126, 127] have found limited application in architecture optimization. In these cases, architectural choices are encoded as discrete design variables which can then be used by optimization algorithms. Repair operators are used to correct invalid combinations, ensuring that architectural constraints are satisfied. Apaza and Selva present the Architecture Decision Diagram (ADD), which enables the user to model architectural decisions [122] based on architecture decision patterns [128], which are then encoded as design variables and dedicated search and repair operators for use in evolutionary algorithms [80, 129]. There exists no architecture optimization method, however:

- that captures all relevant information about the architectures, such as function and form, and characterization of form;
- that supports all types of architectural choices: function decomposition, function-to-form mapping, function and form characterization and specialization, and form connection choices;
- that supports continuous design variables for defining architecture-specific design parameters; and
- that enables automatically *encoding* the problem definition as a numerical optimization problem, and automatically *decoding* design vectors generated by the optimization algorithm into architecture instances.

Observations:

Selective generation of architecture instances by applying optimization techniques prevents the need for full enumeration, which would be infeasible for large design spaces.

Science Gaps:

Currently, no method for formulating SAO problems exists that supports:

- *Definition based on system functions;*
- *Defining architectural choices related to function decomposition, function-to-component allocation, function and component characterization and specialization, and component connection;*
- *Defining architecture-specific design parameters, discrete or continuous;*
- *Automatically encoding the SAO problem as a numerical optimization problem; and*
- *Automatically decoding design vectors into architecture instances.*

1.5.3. Architecture Optimization Problem Characteristics

Optimization represents the automation of a design task: the goal is to find one or more design vectors $\boldsymbol{x}$ that minimize[3] one or more objective functions $f(\boldsymbol{x})$, while satisfying design constraints $g(\boldsymbol{x}) \leq 0$ [49]. Design vectors $\boldsymbol{x}$ are constructed by assigning a value to each of the design variables x; each design vector represents a design point in the design space. In the context of System Architecture Optimization (SAO), a design vector $\boldsymbol{x}$ represents an architecture instance in the architecture design space. The solution space represents the objective and constraint functions evaluated for associated design vectors. A discussion of characteristics of the design and solution spaces for SAO problems follows.

Design Space Characteristics

Mixed-Discrete Variables Design variables are defined from architecture design choices (architectural choices and architecture-specific design parameters) as specified by the architecture design space model. The design variables are *mixed-discrete*, since both continuous and discrete design variables can be part of the problem [130, 131]:

- *Continuous* design variables x_c can take any value between some lower bound $\underline{x}_c$ and some upper bound $\overline{x}_c$ (inclusive). Examples of continuous design variables are wing sweep and engine bypass ratio.
- *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 design variables can be of the following types:

[3]Maximization of an objective function is enabled by negating the objective function value, and therefore minimization is treated as the default in the rest of this work.

- Integer, which takes an integer value between two bounds (inclusive).

  An example of an integer discrete variable is the number of seat rows in an aircraft cabin (e.g. between 20 and 40).
- Ordinal, which takes any value from a specified list of values, where the order of the values is relevant.

  An example of an ordinal discrete variable is the number of engines for an aircraft (e.g. 1, 2 or 4; order is relevant).
- Categorical, which takes any value from a specified list of values, where the order of the values is irrelevant.

  An example of a categorical discrete variable is aircraft power source (there is no order between kerosene, hydrogen and electricity).

Architectural choices are always of discrete type; architecture-specific design parameters can be either discrete or continuous.

Hierarchical Variables SAO design spaces feature strong interaction between design variables as defined from architecture design choices. For example, consider the Apollo mission architecture problem analyzed by SIMMONS [119]: the choice 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. Another example is the launch vehicle design problem by PELAMATTI ET AL. [132], where the number of stages is a choice (1 to 3), as well as several choices such as fuel type and dimensions for each of the stages: it follows that if a one-stage architecture is selected, the choices regarding the second and third stages do not have to be considered.

In general, choices constraining available options of other choices is a common theme in architecture optimization. These interactions between design variables create a *hierarchical* structure in the design space through the following two mechanisms [133]:

- *Activation*: design variables higher up in the hierarchy determining which variables lower in the hierarchy are *active* or *inactive*.
- *Value constraints*: design variables higher up in the hierarchy determining which options are available for lower-level design variables.

Figure 1.9 shows an example design space containing both types of hierarchy. It shows that the Generate Thrust variable determines whether the Generate Torque, Compressor Stages, and Bypass Ratio variables are activated based on its selected value (Turbofan or Propeller). It also defines a value constraint, preventing Batteries from being selected as the Energy Carrier if a Turbofan is selected. The Compressor Stages variable can be activated either by Generate Thrust (if a Turbofan is selected) or by Generate Torque (if a Turboshaft is selected). The selection of the value for Generate Torque additionally constraints the available options for the Energy Carrier through two value constraints. A more detailed discussion of the two mechanisms follows.

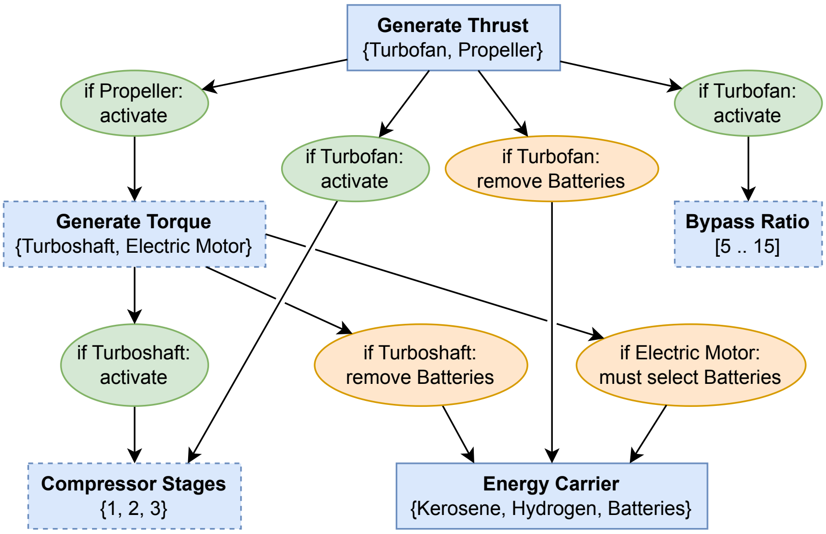


Figure 1.9: Illustration of design variable hierarchy. The blue rectangles represent design variables and their options (discrete options shown in curly brackets; continuous bounds by square brackets). Ellipses represent hierarchical relationships between design variables: green represents activation relationships; orange represents value constraints. Design variables with dashed borders are conditionally active.

Activation Relationships and Imputation An *activation relationship* between two design variables defines for which option selection(s) of the first design variable, the second design variable is activated. Figure 1.9 shows activation relationships as green ellipses.

The property of being active or inactive is called *activeness*, and can be queried through the activeness function $\delta_i(\boldsymbol{x})$ [134], 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*, shown in Figure 1.9 as design variables with dashed borders. 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 [133]. Hierarchical design spaces are also known as conditional design spaces [135] (as design variables can be conditionally active), design spaces with tree-structured dependencies [122, 136, 137], and variable-size design spaces [138, 139] (as inactive design variables can also be seen as the results 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. This wastes computational resources, because the optimizer might require the evaluation of effectively the same design point multiple times. 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 [133]. The process of replacing inactive design variables with canonical values is called design vector *imputation* [133, 140], and the resulting design vector is called an imputed or canonical design vector. The effect on optimizer performance then is twofold:

- Evaluation results (objective and constraint values) of a canonical design vector can be stored. These stored results can then be returned for repeated evaluation requests of the same canonical design vector (possibly originating from a different non-canonical design vector), thereby saving computational resources.
- The optimizer will maintain a one-to-one mapping from design to solution space. This prevents multiple design vectors mapping to the same objective and constraint values, which would result in "flat" areas in the solution space.

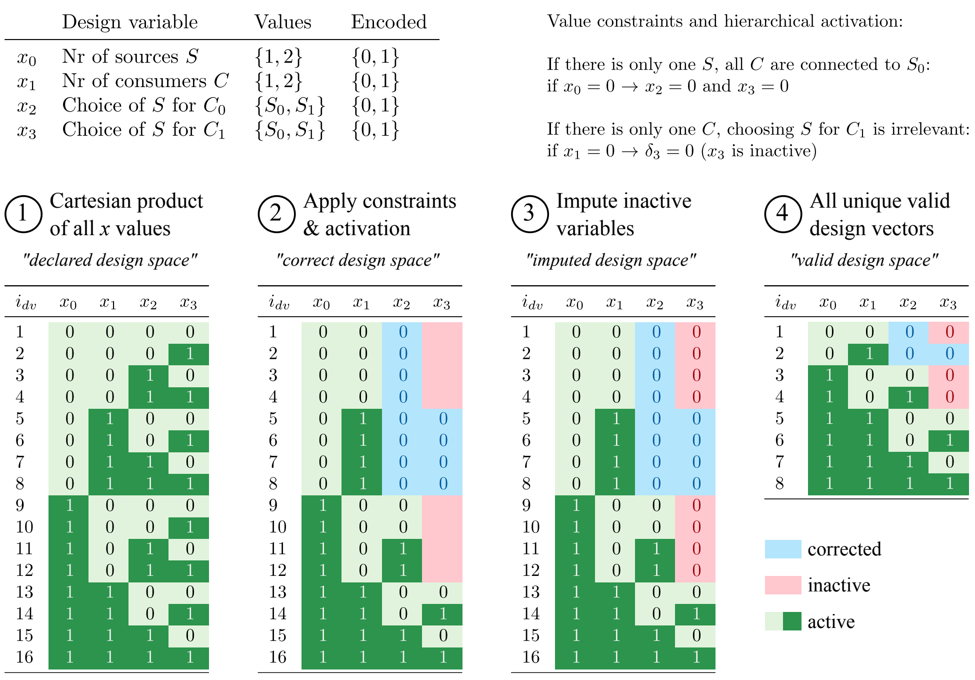
Value Constraints and Correction Restricting option availability of active variables lower in the hierarchy is done using *value constraints*, also known as enumeration constraints in the context of architecture optimization [80] or configuration constraints in product line engineering [62, 69]. Figure 1.9 shows value constraints as orange ellipses. According to the taxonomy by LE DIGABEL & WILD [141] value constraints are:

- Non-quantifiable: it is only known 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.
- Known: they are defined in the design space.

The process of ensuring value constraints are satisfied is called *correction*. A correct and canonical (imputed) design vector is called a *valid* design vector. Since value constraints only depend on the design space definition, correction can be applied as a standalone operation (i.e. without running an evaluation).

Figure 1.10 visualizes the processes of correction and imputation for an example hierarchical design space. It shows a combination of four discrete variables (two options each, shown between brackets) with one conditionally active design variable (x_3 is inactive if $x_1 = 0$) and two value constraints (if $x_0 = 0$, $x_2 = 0$ and $x_3 = 0$). The correction and imputation process goes through four stages:

1. The *declared* design space is defined by the Cartesian product of these four discrete variables resulting in 16 design vectors.
2. Value constraints are corrected for to obtain the *correct* design space.
3. Determining which variables are inactive and imputing their values yields the *imputed* design space.
4. By removing duplicate design vectors (therefore only keeping unique design vectors), the *valid* design space is obtained. The valid design space only consists of 8 design vectors, showing the discrepancy between the sizes of the declared and valid design spaces.



	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

🟩 active

Figure 1.10: 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. The valid design space (step 4) consists of all unique corrected and imputed design vectors.

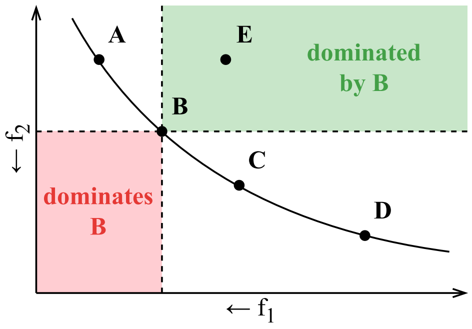
Solution Space Characteristics

Black-Box, Expensive Evaluation Functions The objective and design constraint functions $f(\boldsymbol{x})$ and $g(\boldsymbol{x})$, also known as the *evaluation functions*, are assumed to be *black-box* functions: their behavior is not known in advance. As a consequence, the evaluation functions may 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) [39]. Evaluation functions are assumed to be deterministic: repeated function calls with the same inputs yield the same outputs.

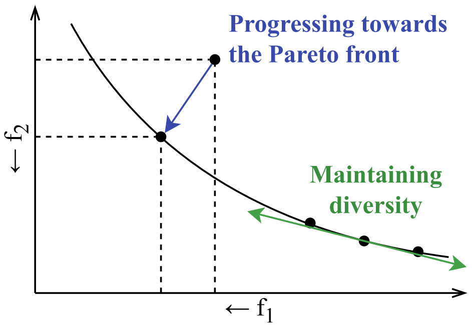
In many cases the evaluation is *expensive* in terms of time and/or computational resources, due to the use of simulation-based evaluation and Multidisciplinary Design Analysis and Optimization (MDAO [72]) approaches. The consequence of this is that the time to perform one evaluation can be orders of magnitude more than the time for the optimization algorithm to generate a new design vector to evaluate. Therefore, to reduce the computational time needed to find an optimum, the evaluation function should be accelerated (e.g. by surrogate modeling or enabling parallel calculations) and/or the number of evaluations requested by the optimization algorithm should be reduced. Since the optimization algorithm can only influence the latter, the focus of this work lies on the reduction of the number of function evaluations N_{fe} needed for convergence [142].

Multiple Objectives When designing a system architecture, needs of system stakeholders may conflict with each other [39] and the associated architecting problem thus becomes a trade-off between these conflicting needs. In general, therefore, an SAO problem is a *multi-objective* optimization problem [79].

Multi-objective optimization results in a set of Pareto-optimal design points [143] rather than a single optimum. This so-called Pareto set is comprised of design points that are not dominating each other (see Figure 1.11a: 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 is that within the Pareto set, no design point is better than any other design point, and that improving one or more of the objective values by moving from one point to another, will necessarily 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 design point diversity along the Pareto front [144], see Figure 1.11b.



(a) Visualization of Pareto dominance relations: B dominates E; A, B, C and D are non-dominated and form the Pareto front.



(b) Challenges of multi-objective optimization problems: progress towards the Pareto front and maintain diversity along the Pareto-front. Figure adapted from [144].

Figure 1.11: Pareto dominance relations and associated challenges of multi-objective optimization algorithms.

Design Constraints SAO problems can, next to value constraints, be constrained by design constraints. *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.

In this work, only *inequality* design constraints are considered; equality constraints can also be defined as two inequality constraints [145], can be eliminated by design variable substitution, or can be eliminated by non-linearly solving implicit residual equations [49]. According to the taxonomy of LE DIGABEL & WILD, design constraints are [141]:

- 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.
- 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.

Hidden Constraints Simulations used for evaluating architecture performance can fail, for example due to an unstable system of equations, infeasible underlying physics, or infeasible geometric parameterization [146]. Any problem that employs simulation could therefore include a so-called *hidden constraint* (also known as unspecified, unknown, forgotten, virtual, and crash constraints [141, 147]): 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. In the taxonomy of LE DIGABEL & WILD [141] hidden constraints are classified as:

- Non-quantifiable: the degree of constraint violation is not available, only whether it is violated or not.
- Unrelaxable: violating the constraint yields no meaningful information about the design space.
- Simulation: a simulation must be run in order to find out the status.
- Hidden: the existence of the constraint is not known before solving the problem.

It is assumed 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, if not longer [147]. Points with violated hidden constraints are denoted *failed*; points where this is not the case are *viable*. The region of failed points can span a large part of the design space: KRENGEL & HEPPERLE report a Failure Rate (FR) of up to 60% for a wing design problem [148], and for an airfoil design problem FORRESTER ET AL. report an FR of up to 82% [146].

PROBLEM FORMULATION

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

$$\begin{array}{clll}
\text{minimize} & f_m(\boldsymbol{x}, \delta(\boldsymbol{x})) & m = 1, 2, \ldots, n_f & \\
\text{w.r.t.} & \underline{x}_{c,i} \leq x_{c,i} \leq \overline{x}_{c,i} & i = 1, 2, \ldots, n_{x_c} & \\
 & x_{d,j} \in \{0, .., N_j - 1\} & j = 1, 2, \ldots, n_{x_d} & \\
\text{subject to} & g_k(\boldsymbol{x}, \delta(\boldsymbol{x})) \leq 0 & k = 1, 2, \ldots, n_g & \\
 & c_{v,l}(\boldsymbol{x}) = 0 & l = 1, 2, \ldots, n_{c_v} & \\
 & c_h(\boldsymbol{x}) = 0 & &
\end{array} \tag{1.1}$$

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 value and hidden constraints (both unrelaxable) 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 explained in the previous sections and summarized in Table 1.2.

Table 1.2: Possible statuses of design points. In order for any of the conditions to be met, conditions and operations above have to be met and performed as well.

	Status if condition is ...		Performed
Condition	met	not met	operations
Any $\boldsymbol{x}$ within bounds	Declared		
Value constraints satisfied	Correct	Invalid	Correction
$\boldsymbol{x}$ is canonical	Valid	Non-canonical	Imputation
Hidden constraint satisfied	Viable	Failed	Evaluation
Design constraint satisfied	Feasible	Infeasible	Optimization
Optimality achieved	Optimal	Non-optimal	Optimization

Observations:

- *SAO problems are characterized by mixed-discrete design variables, design variable hierarchy, and the presence of multiple conflicting objectives.*
- *Due to the need for simulation-based, multidisciplinary evaluation, the evaluation functions may additionally be black-box and expensive, and the problems may include design constraints and hidden constraints.*

1.5.4. OPTIMIZATION ALGORITHMS FOR SAO

The characteristics of SAO problems have several consequences, also summarized in Table 1.3, on the types of optimization algorithms that can be used 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 (due to mixed-discrete and hierarchical variables and black-box nature of the evaluation function) necessitates a global, gradient-free optimization algorithm [49]. Hierarchy induces the need to move design points from the declared to the valid design space by applying correction and imputation [133]. The expensive nature of the evaluation functions drives the need to minimize the number of evaluations needed to find the optimum Pareto set [142], which is challenging for gradient-free algorithms [49]. Finally, constraint handling is needed for both the design constraints [49] and hidden constraints [147].

Many classes of global optimization algorithms exist, designed for various purposes [49]. Exact optimization methods such as grid search [149] or DIRECT [150] are designed to yield reproducible results with provable convergence behavior, and are relevant for problems where finding the true optimum is important [151], however exact methods struggle to solve high-dimensional problems and might need many function evaluations to converge [150]. 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, although without providing mathematical proof of convergence to the true optimum [152].

Table 1.3: Properties of architecture optimization problems and algorithm capability needs.

Space	Property	Capability needs
Design space	Mixed-discrete	Gradient-free optimization [49]
	Hierarchical	Correction and imputation [133]
Solution space	Black-box	Global, gradient-free optimization [49]
	Expensive	Minimize function evaluations [142]
	Multi-objective	Find the Pareto-set [143]
	Design constraints	Constraint handling [49]
	Hidden constraints	Failed area avoidance [147]

Evolutionary Algorithms

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 [153]. Major variations between evolutionary algorithms lie in the way design points are encoded (i.e. the encoding grammar [80]), 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 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 [153] or CMA-ES [154]. Design constraints are handled either by applying a penalty on the objective functions or by integrating constraints in the selection and survival operators directly [153], by reducing the chance of selection/survival for design points with violated constraints.

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 [153]. One of the most popular MOEAs is the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [155] due to its low configuration effort and good performance. NSGA-II uses the non-dominated sorting procedure to select parents for offspring generation, visualized in Figure 1.12. Non-dominated sorting sorts by Pareto-rank first, and within a rank it sorts by crowding distance. Pareto-rank represents which Pareto front the design point belongs to, assigning rank 1 to the overall Pareto front, and subsequent ranks to Pareto fronts obtained by removing design points in the higher-ranking sets. Points are sorted from high to low rank (i.e. low to high rank number), thus meaning that points located closer to the overall Pareto front are preferred. Crowding distance represents the distance to neighboring points within a Pareto rank. Points are sorted from high to low crowding distance, therefore putting closely-spaced points at a disadvantage which results in more diversity along a Pareto front.

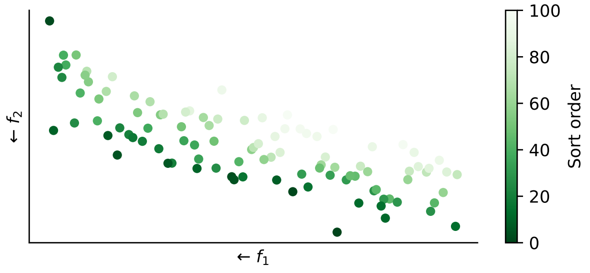


Figure 1.12: Design points sorted by the non-dominated sorting procedure, resulting in points closer to the Pareto front (towards the left bottom of the figure) being ordered before points further away.

Finally, EAs can explore hierarchical design spaces using the hidden genes approach [138, 139, 156–159], where genes representing inactive design variables are retained in the chromosome, however are hidden since they are not "expressed" in the evaluation function. EAs additionally support hidden constraints using the extreme barrier approach [160], where objective and constraint values of failed evaluations are set to $+\infty$ and therefore have less chance to be selected for generating offspring for subsequent generations. The appropriateness of evolutionary algorithms in general and NSGA-II in particular for SAO is noted by CRAWLEY ET AL. [39] and has been demonstrated in the past by various applications, see Table 1.4. Evolutionary algorithms have also found application for optimizing software architecture [161] and software product lines [124], which involves optimization problems characterized by strong hierarchy and choice dependencies, similarly to SAO.

As powerful as evolutionary algorithms are for dealing with the challenges of architecture optimization, the high number of needed function evaluations N_{fe} [165] poses a problem if the objective and constraint functions are expensive to evaluate. As can be seen in Table 1.4, N_{fe} typically is in the order of thousands to hundreds of thousands for EA.

SURROGATE-BASED OPTIMIZATION ALGORITHMS

To solve problems with expensive evaluation functions, *Surrogate-based Optimization* (SBO) algorithms have been developed [142, 166]. SBO algorithms use surrogate models (also known as response surface models or metamodels): models that approximate some expensive function by fitting some mathematical function(s) to training data generated from the expensive function. Evaluating a surrogate model is computationally very cheap compared to the fitted function. During the SBO process, surrogate models are created to approximate the objective and constraint values as a function of the design

Table 1.4: Past applications of evolutionary algorithms to architecture optimization problems. Nomenclature: $n_{\text{valid,discr}}$ = number of valid discrete design points, n_{x_c} = number of continuous dimensions, n_f = number of objectives, n_g = number of constraints, N_{fe} = number of function evaluations.

Algorithm(s)	Application	$n_{\text{valid,discr}}$	n_{x_c}	n_f	n_g	N_{fe}	Ref.
GA[1]	Supersonic aircraft	576	100	10		200 000	[162]
NSGA-II	Aircraft engine	1 163	30	3	15 [2]	4 000	[11]
ACO + GA	Aircraft subsystem	9 600		2		96 000	[79]
NSGA-II	Aircraft family	21 875	4	4		200 000	[163]
NSGA-II	Launch vehicle	123 000	23	3		50 000	[130]
NSGA-II	Space transport	49e6	30	2	4	50 000	[164]
MOEA	GN&C system	79e6		2		1 000	[122]
([3])	Satellite instruments	8.8e12		4		20 000	[80]

[1] Acronyms: 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).
[2] This problem additionally featured hidden constraints.
[3] A special-purpose evolutionary algorithm for architecture optimization was used.

variables. These surrogate models are then used to efficiently determine interesting extra design point(s) to evaluate: the *infill* point(s). Infill points are selected using infill criteria (also known as acquisition functions) evaluated on the surrogate model, which are normally defined to represent some kind of trade-off between exploration (finding new interesting regions in the design space) and exploitation (improving already interesting regions in the design space). 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. Figure 1.13 visualizes the SBO process.

The main choices involved in using SBO are the initial sampling (Design of Experiments) method for training the first surrogate model, the type of surrogate model, the infill criterion, and the termination criterion. Many sampling methods for creating the initial Design of Experiments (DoE) have been developed in the past [49]; the most popular methods for SBO are random sampling, low-discrepancy sequences (e.g. Sobol sequences) or Latin Hypercube Sampling (LHS) [167].

Especially powerful types of SBO algorithms are *Bayesian Optimization* (BO) algorithms [167]. BO algorithms use surrogate models that provide standard error estimates $\hat{\sigma}(\boldsymbol{x})$ 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) surrogate models [165, 168], also known as Kriging models [169]. One of the most popular BO algorithms is Efficient Global Optimization (EGO), which uses GP models and the Expected Improvement (EI) infill criterion to solve continuous, unconstrained, single-objective optimization problems [142].

Extensive research has been performed on handling design constraints [170, 171] and solving multi-objective [172, 173] problems using BO. GP models were originally

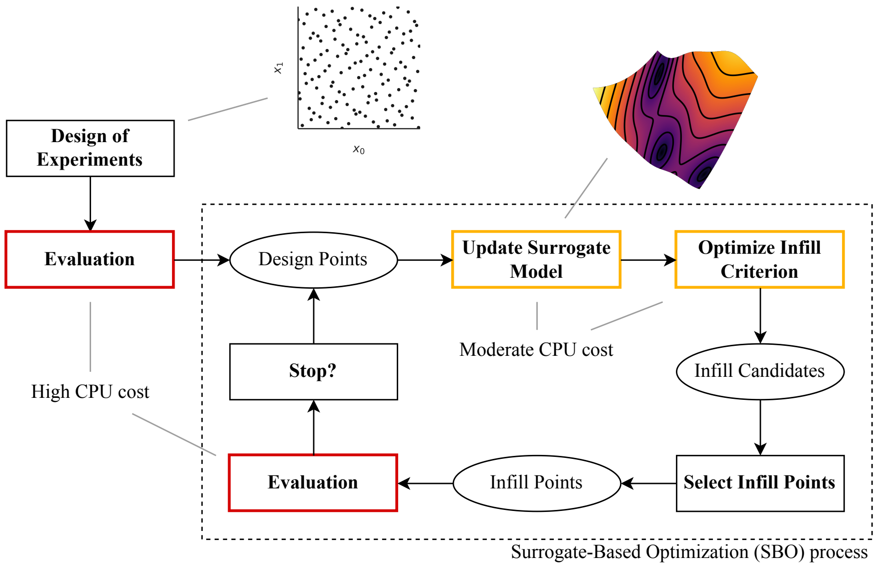


Figure 1.13: The Surrogate-Based Optimization (SBO) process.

developed for continuous variables, however research has been performed into methods for supporting mixed-discrete [174–180] and hierarchical [26, 132, 133, 137, 181–183] design spaces. Enabling the use of GP models for high-dimensional design spaces is an active area of research [167, 184–186]. BO has also been extended with strategies for dealing with hidden constraints [147]. BO has not been used to solve SAO problems, and although BO has been applied for some combinations of SAO problem challenges, it has not been demonstrated for a combination of all challenges. Especially the combination of a mixed-discrete, hierarchical design space with hidden constraints has not yet been explored.

ADDITIONAL CONSIDERATIONS

Hierarchical design spaces introduce three additional considerations for global optimization algorithms:

- Due to the hierarchical nature of the SAO design space, global optimization algorithms can either be applied in a *global approach* or in a *decomposition-based approach* [139]. The decomposition-based approach decomposes the hierarchical design space into multiple non-hierarchical design spaces which are then solved individually, as for example done in [115, 130]. The global approach, on the other hand, maintains the hierarchical structure of the design space and considers this behavior directly in the optimization algorithm (e.g. by correcting and imputing design vectors).

  Compared to the global approach, the decomposition-based approach requires the definition of two additional heuristics [132, 139]: how to decompose the problem and how to assign function evaluation budget (N_{fe}) to each of the decompositions. Additionally, evidence shows that the global approach leads to better optimizer performance (i.e. how closely the optimum can be approached) for a given function evaluation budget, both for MOEA [157–159, 187] and for BO [132, 187, 188]. For these reasons, only the global approach is considered in this work.

- When *sampling* the design space to create the initial DoE, some regions in the design space may be over- or under-represented due to hierarchy in the design space [39]: a region in the design space that has inactive variables contains less valid design vectors than a region with more active variables. Existing sampling algorithms do not take this effect into account [49].
- Design variable *correction* might affect design space exploration by modifying design vectors generated by the optimization algorithm. It is not clear what the impact is of the correction algorithm on the performance of optimization algorithms.

Observations:

- *Evolutionary algorithms and Surrogate-Based Optimization (SBO), in particular Bayesian Optimization (BO), algorithms are powerful classes of global, gradient-free optimization algorithms.*
- *BO can solve optimization problems with less function evaluations than evolutionary algorithms.*

Science Gaps:

- *Existing sampling algorithms do not explicitly take the hierarchical nature of the design space into account.*
- *It is not clear what the impact is of the selection of the correction algorithm on the performance of optimization algorithms.*
- *BO has not been demonstrated for the combination of all previously-mentioned SAO problem characteristics: mixed-discrete, hierarchical, multi-objective, constrained by design and hidden constraints, expensive, black-box optimization.*

1.5.5. EVALUATING ARCHITECTURE INSTANCES

The architecture evaluator represents the evaluation function in an optimization process: it returns performance metrics for a given architecture instance, thereby providing the optimization algorithm with feedback for searching the design space. The evaluation function is problem-specific and can be implemented as anything ranging from a simple script using lookup tables or low-fidelity handbook equations, to distributed multi-disciplinary high-fidelity simulation toolchains, as long as it fulfills the following requirements [119]:

- The performance metrics should be *sensitive* to all relevant architecture design choices. If architecture design choices do not influence any of the performance metrics, they should either be removed from the design space or the evaluation function should be modified to include effects due to these choices.
- The performance metrics should be *available* for all possible architectures, and with similar accuracy/fidelity. This ensures that the evaluation function is flexible enough to handle all possible architectures, and that architecture instances can be fairly compared to each other. Metrics used as objectives f should always be available; metrics used as constraints g may not always be available for a given architecture instance.
- The evaluation function should be executable *without user interaction* when running the optimization. This ensures that the design space can be searched automatically.

Some research has been performed on methodologies for integrating system architecture models with system simulation, to ensure consistency between the system architecture and evaluation code and to enable verification and validation [53]. Directly integrating calculations in architecture models is for example possible using MAXIM in OPM [58], priced feature models [125], SysML parametric diagrams [54], SysPhs SysML extension [189], or SysMLv2 expressions [90]. Direct integration means that calculations as needed for performance evaluation are specified in the same model as the system architecture, for example using component properties as inputs and calculating performance metrics using relatively simple equations. System architecture modeling languages and/or tools, however, are not appropriate for more complicated computations. Rather, connections to external simulation environments should be established in those cases, for example Modelica [190, 191], Knowledge-Based Engineering (KBE) tools [192], or dedicated solvers [193, 194].

Simulating complex systems at the system-level often involves multiple interacting heterogeneous system models [36]. One way to simulate such systems is using co-simulation, a set of methods and tools for solving coupled systems of equations, both in discrete and in time domains [195]. Standardized interfaces such as the Functional Mockup Interface (FMI) [196, 197] enable model exchange between design teams involved in designing a system.

(COLLABORATIVE) MULTIDISCIPLINARY DESIGN ANALYSIS AND OPTIMIZATION

As previously discussed in Section 1.4, MDAO is another useful method for obtaining system-level performance metrics by numerically coupling disciplinary analysis tools and ensuring all computations are consistent with each other [72]. This enables integrating calculations stemming from all relevant engineering disciplines, none of which can be ignored or should be allowed to take the overhand in defining the design. It fits well with systems engineering, as there the goal is also to integrate all relevant engineering disciplines into one system-level design (see Section 1.2), and using MDAO leads to better performing systems that are designed in less time when compared to sequential design approaches [49]. For simulation-based system architecture performance evaluation, MDAO is considered a key enabler [53], as is also shown by recent interest in coupling MBSE to MDAO [77, 198–205]. However, changes in system architecture were not synchronized to the MDAO workflow, rendering these approaches not usable for SAO.

Local integration and execution of MDAO workflows is difficult or infeasible for the distributed and/or diverse disciplinary competences and tools needed to design a complex system. *Collaborative MDAO* allows coupling disciplinary analysis tools that are developed and managed by different teams, potentially from different organizations, in a single MDAO workflow [36], visualized in Figure 1.14. It is characterized by the application of a Central Data Schema (CDS) and the distributed, cross-organizational management, development, and execution of analysis tools.

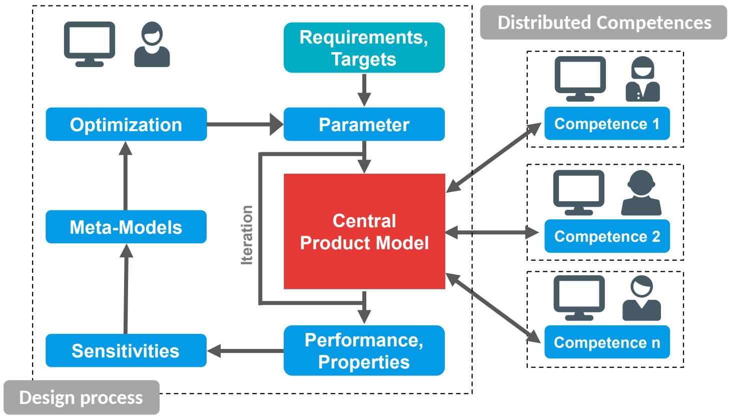


Figure 1.14: Principles of collaborative MDAO, showing the distributed management, development and execution of disciplinary competences, integrated through a central product model. Figure adapted from [206].

The application of a CDS is essential to ensure all tools "speak the same language" [207]: the CDS defines the common vocabulary and data storage structure of the central product model (see Figure 1.14), from (to) which each tool reads (writes) its inputs (outputs). This ensures data syntax and semantics are consistent for tools originating from diverse engineering disciplines. It also reduces the amount of interfaces to implement from $N(N-1)$ (quadratic growth; with N being the number of disciplinary analysis tools) if tool-to-tool ad-hoc data exchange is implemented, to $2N$ (linear growth) if a CDS is used [207], as also shown in Figure 1.15. Data interfaces are implemented by the tool developers, shifting this burden away from the workflow integrator, who can focus on setting up and running the workflow.

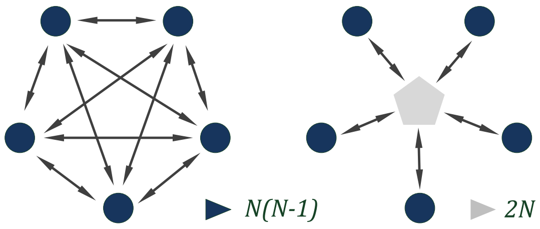


Figure 1.15: Visualization of the reduction in data interfaces from $N(N-1)$ to $2N$ when using a Central Data Schema (CDS). Figure adapted from [207].

The distributed, cross-organizational execution of collaborative MDAO workflows needs to comply with network security and intellectual property right constraints imposed by the collaborating organizations [36]. This enables disciplinary tools to be used in MDAO systems outside of the developing organization, without sharing the tool and while keeping control over computational resources. Collaborative MDAO problems are executed in Process Integration and Design Optimization (PIDO) environments, which handle aspects like cross-platform disciplinary tool integration and sharing, data orchestration, execution status monitoring, and distributed (cross-organizational) execution [208]. Before a workflow can be executed, some of these features may require manual configuration. Although workflows can be defined directly in PIDO tools, a more formalized approach, based on a CDS that enables automatic data matching and MDAO problem definition, requires specialized MDAO formulation tools [19, 209]. Such formulation tools can automatically connect inputs and outputs between the different tools, determine execution order, and (with some amount of user input) make the MDAO problem executable by adding solvers and/or applying an MDAO architecture [210]. Formulation tools enhance insight into what data is generated and consumed at what point in the workflow, which can support discovering and solving bugs in disciplinary tools. Therefore, there is usually a clear separation between the MDAO problem formulation phase and the execution phase [211], as shown in Figure 1.16.

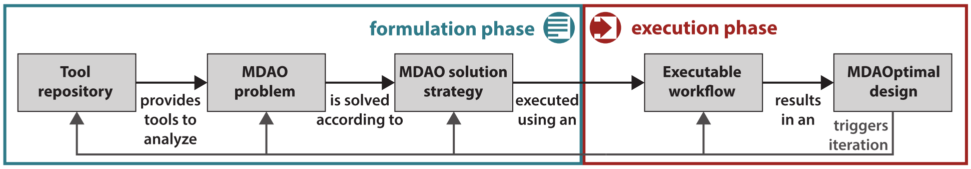


Figure 1.16: The main steps and interactions involved in executing a collaborative MDAO workflow, showing the formulation and execution phases. Figure adapted from [208].

Collaborative MDAO is a powerful methodology to design complex systems in large project consortia, and should therefore be available as a way to evaluate architecture instances in SAO. Effort towards applying collaborative MDAO for SAO is relatively recent. HELLE ET AL. [212] present a method for modeling variability-aware analysis architectures in SysML using Parametric Analysis Models (PAM), and executing analysis architectures from architecture instances manually defined from a superset model. Bruggeman et al. [213] present an MDAO workflow that dynamically swaps a sub-workflow depending on a selection of the manufacturing process of the part being designed. Sonneveld et al. [214] demonstrate an MDAO workflow that dynamically modifies a sub-problem: the number of design variables governing the skin thickness is not known in advance, however is determined based on the number of ribs selected in an aileron. The integration of these methods into a collaborative MBSE and MDAO framework is presented in [78].

As discussed before, an architecture evaluator should be flexible enough to handle all architectures, it should be sensitive to relevant architecture design choices, and it should be executable without user interaction. Translated to collaborative MDAO, this means that:

- To be flexible enough to handle all architectures, the execution behavior (i.e. data connections and tool inclusion and execution order) of the collaborative MDAO workflow should be adapted to the architecture instance being evaluated. Existing methods do not fully support changing MDAO workflow behavior due to changes in system architecture.
- To be sensitive to relevant architecture design choices, all relevant data from the architecture instance (e.g. function allocation, component selection and characterization, component connections) should be communicated to the MDAO workflow. Existing MBSE-to-MDAO methods do not transfer all relevant architecture data [53]: changes in system architecture can therefore not be communicated, or only in a limited manner.
- To be executable without user interaction, the architecture generator and the collaborative MDAO workflow should be executed in a shared computational environment. This has not been demonstrated before.

Observations:

- *Defining system-level performance metrics that are sensitive to architecture design choices enables comparison of architecture instances.*
- *Collaborative MDAO supports calculating metrics that requires solving coupled computational problems, where different computational blocks are provided by cross-organizational heterogeneous teams of experts.*

Science Gaps:

- *MDAO workflows currently do not fully support dynamically changing their behavior due to changes in system architecture.*
- *Existing MBSE to MDAO connection methods do not support transferring all relevant architecture data to the collaborative MDAO workflow.*
- *Generating architectures in the same computational environment as where the MDAO workflow is executed has not been demonstrated before.*

1.6. Research Question and Objectives

Based on the observations in the previous sections, the main research question is as follows:

> How can System Architecture Optimization (SAO) improve the design space exploration and optimization of complex systems?

The research question is divided into three sub-questions:

1. How to formulate SAO problems based on system functions?
2. How to solve SAO problems using global optimization algorithms?
3. How to evaluate and optimize system architecture instances in a collaborative MDAO environment?

To answer the research questions, the objective of this research is

> to enable practical usage of System Architecture Optimization (SAO), by developing a methodology for formulating SAO problems, providing algorithms for solving these problems, and by using collaborative MDAO to evaluate architecture instances.

The research objective is broken into three sub-objectives:

1. Develop global evolutionary and Bayesian Optimization (BO) algorithms capable of efficiently solving SAO problems, by:
    (a) integrating information about the hierarchical design space into the optimization algorithms;
    (b) developing a sampling algorithm that explicitly takes the hierarchical nature of the design space into account;
    (c) investigating the impact of the correction algorithm on optimization performance; and
    (d) developing a strategy for solving problems with hidden constraints when using BO algorithms.
2. Develop a way to formulate SAO problems, consisting of:
    (a) a method for modeling SAO problems based on system functions, that supports the definition of architectural choices and architecture-specific design parameters; and
    (b) algorithms to encode the model as an optimization problem (i.e. in terms of design variables, objectives, and constraints), and to decode design vectors into architecture instances.
3. Develop a methodology for leveraging collaborative MDAO for quantitative architecture evaluation, by:
    (a) enabling changing MDAO workflow behavior due to architectural changes;

(b) propagating architecture data to the collaborative MDAO workflow; and
(c) integrating SAO within a collaborative MDAO process integration platform.

The research sub-objectives presented above can be mapped to the optimization process according to MARTINS & NING [49], as shown in Figure 1.17: formulating the SAO problem, optimization algorithms ("update design variables"), and using collaborative MDAO to "evaluate objective and constraints". This is also how this dissertation is structured: Chapter 2 proposes improvements to efficient optimization algorithms for SAO (sub-objective 1). The architecture design space modeling and problem formulation methodology (sub-objective 2) is presented in Chapter 3. Integration with collaborative MDAO (sub-objective 3) is presented in Chapter 4. Chapter 5 concludes the dissertation and provides recommendations for future research.

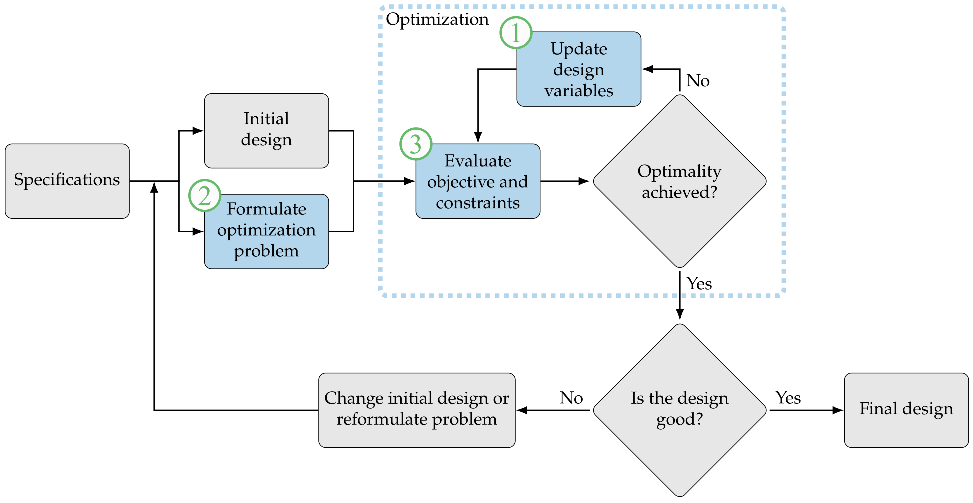


Figure 1.17: Design optimization process according to MARTINS & NING [49], with the three research sub-objectives highlighted. Figure adapted from [49].

2

Optimization Algorithms

Chapter Contents

This chapter presents optimization algorithms for solving System Architecture Optimization (SAO) problems. This forms the first part of the objectives presented in Section 1.6.

As discussed in Section 1.5.3, SAO problems feature one or more of the following characteristics: mixed-discrete design variables, design variable hierarchy (conditionally active design variables and value constraints), black-box, expensive evaluation functions, multiple objectives, design constraints, and hidden constraints. The first thing to note is that if the evaluation functions are not expensive to evaluate or if the design space only consists of several architecture instances, it might be feasible to fully enumerate and evaluate the design space. In the general case, however this is not feasible and optimization algorithms will have to be used; this will be the main assumption underlying the remainder of this chapter.

This chapter is based on [3, 4, 6, 7].

As discussed in Section 1.5.4, global optimization algorithms will be used to solve SAO problems, in particular Multi-Objective Evolutionary Algorithms (MOEAs) and Bayesian Optimization (BO) algorithms. One is not better than the other: MOEAs 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) [159]. BO 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 [165].

Section 2.1 presents the non-hierarchical basis of the BO algorithm further developed in this work. Following sections then present the contributions of this work to optimization algorithms for SAO:

- metrics to characterize hierarchy in SAO problems (Section 2.2);
- improvements to optimization in hierarchical design spaces (Section 2.3): test problems, hierarchical sampling and correction, and including hierarchy information into algorithms;
- strategies for dealing with hidden constraints in BO (Section 2.4); and
- the implementation of results in the open-source SBArchOpt library (Section 2.5).

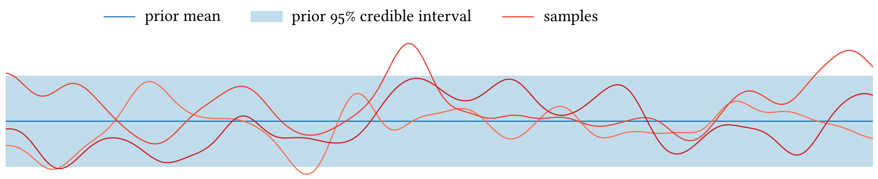
The developments are demonstrated using a jet engine architecture optimization problem in Section 2.6. Section 2.7 concludes the chapter. The dataset containing all experimental results presented in this chapter is available at [18]. The code used to run the experiments is available at [1].

[1]github.com/jbussemaker/ArchitectureOptimizationExperiments (HIDDEN-CONSTRAINTS branch)

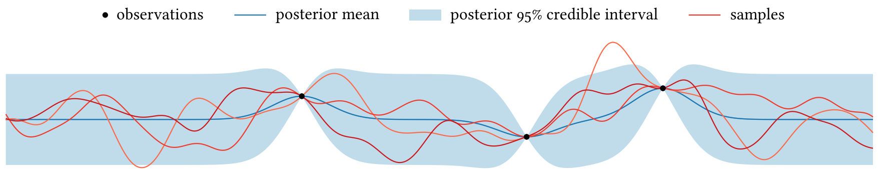
2.1. Non-hierarchical Foundation of the Bayesian Optimization Algorithm

This section presents the non-hierarchical basis of the Bayesian Optimization (BO) algorithm used and further developed in this work. 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 [215]. In this work the same approach is followed: specializations (e.g. hierarchy and hidden constraints support) build on prior specializations (e.g. mixed-discrete and multi-objective support).

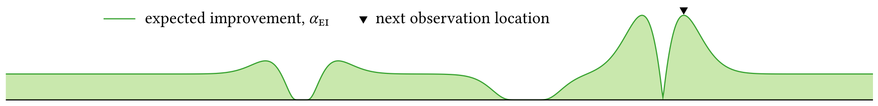
Similarly to most application of BO, the BO algorithm developed in this work uses Gaussian Process (GP) models [165, 168], also known as Kriging models [169]. For each objective f and constraint g, one GP model is trained. GP models work by starting from a prior distribution of estimated values and obtaining a posterior distribution by fitting the prior to observed (sampled) values [168], see Figure 2.1. The prior represents the prior beliefs (i.e. before observation) about the kind of function to be modeled (Figure 2.1a). Because the objective and constraint functions are non-linear black-box functions, no assumptions can be made about their structure, and thus the prior is set to a constant value of 0 with a standard deviation of 1.



(a) Gaussian Process (GP) prior distribution, showing confidence interval and three functions sampled from the process.



(b) Gaussian Process (GP) posterior distribution, updated by three observations, clearly showing the reduced variance near the observed points.



(c) Expected Improvement (EI) function for the above GP posterior, indicating where the next observation should be performed in order to maximize the function. Note that here EI is selected by maximization, and EI is slightly higher at the selected point than at the "hill" to the left of it.

Figure 2.1: Visualization of a Gaussian Process (GP) and the Expected Improvement (EI) infill criterion. Figures reproduced from [167].

The posterior is constructed by a covariance function (also known as kernel) which relates points to each other using some measure of distance (Figure 2.1b). The result of this is that points in the GP that are "closer" to observed points have a value that is "closer" to the observed value as well. Additionally, the variance of the GP is reduced to 0 near observed points, and towards the variance of the prior distribution further away from sampled points. For a given non-observed point $\boldsymbol{x}$, a GP can thus both give the most likely value $\hat{y}(\boldsymbol{x})$ (Eq. (A.4)) of that point and a standard deviation $\hat{\sigma}(\boldsymbol{x})$ (Eq. (A.5)), which represents how confident the model is about the mean value prediction at $\boldsymbol{x}$.

The fact that both $\hat{y}(\boldsymbol{x})$ and $\hat{\sigma}(\boldsymbol{x})$ are available enables the formulation of infill criteria that balance exploration and exploitation without the need for any manual configuration. This is exactly what the popular Expected Improvement (EI) criterion provides in the Efficient Global Optimization (EGO) algorithm [142]: EI($\boldsymbol{x}$) is calculated from the area under (i.e. the integral of) the part of the cumulative distribution function of $\hat{\sigma}(\boldsymbol{x})$ that improves over the best observed f, see Figure 2.2 for a visualization. Points with a higher EI therefore represent points with a higher potential for improving the current best point (Figure 2.1c). This explanation skips over many of the mathematical and practical details, and the interested reader is referred to [167, 168] for more details. [216] provides an interactive visualization of GP modeling.

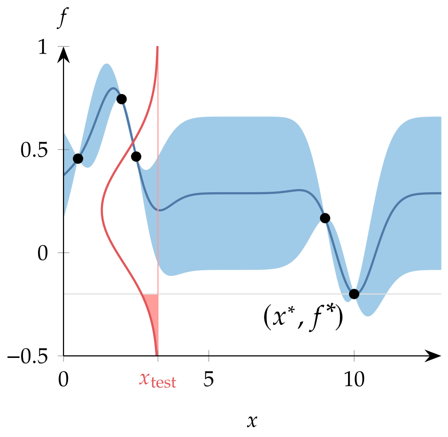


Figure 2.2: Visualization of the Expected Improvement (EI) infill criterion. At x_{test}, the probability distribution is highlighted, including the shaded region that represents an improvement over the current best function value f^*: EI represents the centroid of the shaded region.

The BO algorithm uses a GP model that supports mixed-discrete, including categorical, design variables. 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 integer and categorical variables is based on the work by SAVES ET AL. [174], which uses dedicated kernels to model categorical variables.

Infill points are selected by formulating an infill ensemble. An ensemble combines multiple infill criteria into a multi-objective infill optimization problem, solved using the NSGA-II multi-objective evolutionary algorithm, and infill points are selected from the resulting Pareto front (see Section A.2 for more details). There are two advantages to the ensemble-infill strategy [217]:

1. The selected infill points are a compromise of the underlying infills, thereby mitigating the problem with different infill criteria suggesting to explore very different parts of the design space [218].
2. It is trivial to select multiple infill points per iteration for batch optimization without needing to retrain the underlying GP models [167].

For single-objective problems, an ensemble of Lower Confidence Bound (LCB) [219] (Eq. (A.9)), Expected Improvement (EI) [142] (Eq. (A.10)), and Probability of Improvement (PoI) [220] (Eq. (A.11)) infills is used (see Section A.3). For multi-objective problems, the ensemble consists of the Minimum Probability of Improvement (MPoI) [221] (Eq. (A.12)) and Minimum Euclidean PoI (MEPoI) [10] (Eq. (A.13)) infills (see Section A.4). The infill batch size n_{batch} is set to the maximum amount of designs that can be evaluated in parallel by the architecture evaluation code: $n_{\text{batch}} = n_{\text{parallel}}$. See Appendix A for more details about the infill optimization procedure and the infill ensembles.

Inequality design constraints are handled by constraint function mean prediction $\hat{g}$ [171] (Eq. (A.14)) by default, or by Probability of Feasibility (PoF) [222] (Eq. (A.15)) if a more or less conservative (achieved by PoF above or below 50%, respectively) criterion is requested. To summarize, the non-hierarchical basis of the BO algorithm can deal with the following SAO problem challenges: mixed-discrete design variables, black-box, expensive evaluation functions, multiple objectives, and design constraints. Support for design variable hierarchy and hidden constraints is developed and presented in the following sections.

2.2. HIERARCHICAL DESIGN SPACE CHARACTERIZATION

This section introduces four metrics for characterizing hierarchical design spaces, developed as part of this work and discussed in the following sections:

- *Imputation ratio* IR: the ratio between the size of the declared design space (i.e. only considering the design variable definitions) and the size of the valid design space (i.e. only containing valid design vectors). A higher value means more discrepancy.
- *Correction ratio* CR: the ratio between the size of the declared design space and the size of the correct design space (i.e. the number of correct design vectors considering only value constraints). A higher value means more discrepancy.
- *Correction fraction* CRF: the fraction of hierarchy that is due to the need for correction (correcting value constraints) versus the need for imputation (imputing inactive design variables). The larger this value, the larger the impact of correction.
- *Max rate diversity* MRD: the largest encountered rate diversity in a problem. Rate diversity is a measure for how often the different values of a given discrete design variable are encountered in all valid design vectors. The higher this value, the larger the difference in occurrence rates.

2.2.1. IMPUTATION AND CORRECTION RATIO

Imputation Ratio Due to hierarchy, the valid design space (i.e. the set of all design vectors where value constraints are satisfied and all inactive variables have been imputed) might be much smaller than the declared design space (see Table 1.2), as also discussed in Section 1.5.3. Since this might pose a challenge to optimization algorithms, a metric to quantify this discrepancy is introduced: the ratio between the declared and valid discrete design space sizes is defined as the discrete *imputation ratio* IR_d:

$$\mathrm{IR}_d = \frac{\prod_{j=1}^{n_{x_d}} N_j}{n_{\mathrm{valid,discr}}}, \tag{2.1}$$

where n_{x_d} is the number of discrete design variables, $n_{\mathrm{valid,discr}}$ 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 problem, values higher than 1 indicate hierarchy: for the suborbital vehicle design problem in [130], 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 [223] as used in software product line engineering is the reciprocal of discrete imputation ratio. The continuous imputation ratio IR_c is defined as follows:

$$\mathrm{IR}_c = \frac{n_{\mathrm{valid,discr}}\; n_{x_c}}{\sum_{l=1}^{n_{\mathrm{valid,discr}}} \sum_{i=1}^{n_{x_c}} \delta_i(\boldsymbol{x}_{d,l})}, \tag{2.2}$$

where $\delta_i(\boldsymbol{x}_{d,l})$ is the activeness function for continuous variable i for valid discrete design vector $x_{d,l}$, $n_{\mathrm{valid,discr}}$ 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:

$$\mathrm{IR} = \mathrm{IR}_d\, \mathrm{IR}_c, \tag{2.3}$$

The example problem from Figure 1.10 has 16 declared design vectors (4 design variables with 2 options each) of which only 8 are valid, so it has an imputation ratio of $\mathrm{IR} = \mathrm{IR}_d = 16/8 = 2$.

Correction Ratio Similarly to the discrepancy between the declared and valid design space sizes, the discrepancy between the declared and correct (i.e. the set of design vectors where value constraints are satisfied, however where inactive variables have not been imputed yet; see Table 1.2) design spaces sizes can also be quantified. 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. The *correction ratio* CR is defined as:

$$\mathrm{CR}_d = \frac{\prod_{j=1}^{n_{x_d}} N_j}{n_{\mathrm{corr,discr}}}, \tag{2.4}$$

$$\mathrm{CR}_c = \frac{n_{\mathrm{corr,discr}}\, n_{x_c}}{\sum_{l=1}^{n_{\mathrm{corr,discr}}} \sum_{i=1}^{n_{x_c}} \delta_i(\boldsymbol{x}_{d,l})}, \tag{2.5}$$

$$\mathrm{CR} = \mathrm{CR}_d\, \mathrm{CR}_c, \tag{2.6}$$

where CR_d is the discrete correction ratio, $n_{\mathrm{corr,discr}}$ the number of correct discrete design vectors, and CR_c the continuous correction ratio.

Correction Fraction The impact of the need of correction to the design space hierarchy can be quantified by the *correction fraction* CRF:

$$\mathrm{CRF} = \frac{\log \mathrm{CR}}{\log \mathrm{IR}}, \tag{2.7}$$

CRF varies between 0% and 100%, where 0% indicates no hierarchy is due to correction ($\mathrm{CR} = 1$) and 100% indicates all hierarchy is due to correction ($\mathrm{CR} = \mathrm{IR}$, and there are no activeness relationships).

The valid design vectors shown in Table 2.1 represent a design space with a declared size of 12 ($N_0 \cdot N_1 = 4 \cdot 3 = 12$), however $n_{\mathrm{valid,discr}} = 6$, and therefore $\mathrm{IR} = \mathrm{IR}_d = 12/6 = 2.0$. Additionally, the example has $n_{\mathrm{corr,discr}} = 10$, because the inactive design variables can take any declared value and still represent a correct (but not necessarily valid) design vector. Therefore, $\mathrm{CR} = \mathrm{CR}_d = 12/10 = 1.2$. From this, $\mathrm{CRF} = \log 1.2 / \log 2.0 = 26\%$, indicating that 26% of the design space hierarchy is because of the need for correction, while the other 74% are because the need for imputation of inactive design variables.

Table 2.1: 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	🟥

2.2.2. 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. [39]. 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 possibilities to choose from, however 88% of those possibilities are made up of the subsets composed of 4, 5, and 6 elements. This means that the subset of all other sizes are represented much less in the total number of possibilities. This observation can be extended to design variable values too: for a large version of the engine architecting benchmark problem presented in Section 2.3.1, 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, because the turbofan design space contains more active design variables. Therefore there is a large gap between how often the two possible values of this design variable appear in all valid discrete design vectors $x_{\text{valid,discr}}$. 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 at the problem level:

$$\text{Rates}_j = \left\{\text{Rate}(j,\delta_j=0),\text{Rate}(j,0),..,\text{Rate}(j,N_j-1)\right\}, \tag{2.8}$$

$$\text{RD}_j = \max\text{Rates}_j - \min\text{Rates}_j, \tag{2.9}$$

$$\text{MRD} = \max_{j\in 1,..,n_{x_d}} \text{RD}_j, \tag{2.10}$$

with j the index of the discrete design variable, $\delta_j = 0$ denoting the case when design variable j is inactive, and $\text{Rate}(j,\text{value})$ returning the relative occurrence rate of that value in all valid discrete design vectors $x_{\text{valid,discr}}$. 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 $\text{Rate}(j,\delta_j=0)$ included. If the inactive case is included, the subscript "all" is added to denote all cases and values are considered.

Table 2.2 shows occurrence rates, the rate diversity and maximum rate diversities of a smaller version of the turbofan problem. The rate diversity of the choice between turbojet and turbofan architectures (defined by x_1) is not as extreme as for the aforementioned larger version, however there is still a rate diversity of 60% as only 20% of possible discrete design vectors represent a turbojet architecture.

A large MRD thus indicates that for at least one discrete design variable some of its values occur often in $x_{\text{valid,discr}}$, whereas others occur rarely. This effect should be considered when exploring the design space, to prevent spending not enough computation effort exploring (or even skipping over) the values that occur rarely.

Table 2.2: Rate diversity RD of the simple turbofan architecting problem. The maximum rate diversity MRD values are underlined. Only discrete variables are shown, as rate diversity does not apply to continuous variables.

	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}	$\underline{60\%}$	57.1%	20%	20%	28.6%	28.6%
RD	$\underline{60\%}$	57.1%	0%	0%	15.4%	15.4%

2.3. Optimization in Hierarchical Design Spaces

This section deals with optimization in hierarchical design spaces using MOEAs and BO algorithms. First, several hierarchical test problems are presented in Section 2.3.1. Then, the section presents results of the research performed:

- A hierarchical sampling algorithm for creating a Design of Experiments (DoE) in hierarchical design spaces to deal with max rate diversity (MRD) effects (Section 2.3.2).
- An investigation into problem-agnostic correction algorithms for hierarchical design spaces to deal with correction ratio (CR) effects (Section 2.3.3).
- An investigation into whether including more information about the hierarchical design space (e.g. activeness information and the availability of $x_{\text{valid,discr}}$) into the optimization algorithm is beneficial (Section 2.3.4).

2.3.1. Test Problems

Developing optimization algorithms requires test problems that behave as "realistic" problems would also behave [224]. For SAO, that means that the test problems are based on system architecting activities and that their evaluation code is derived from physics simulations. Evaluation time should be kept near-instantaneous to enable testing many configurations of an optimization algorithm, and allow repeating a test multiple times to correct for randomness in the tested algorithms. Additionally, they should be available openly to enhance transparency and reproducibility of presented optimization results, and they should support tuning the difficulty of the problem [224].

As part of this work, several existing test problems have been modified and used, however three main test problems stand out due to their basis in realistic SAO problems:

- A multi-stage rocket design problem ("Rocket"), which is a mixed-discrete (6 continuous and 8 discrete design variables), multi-objective (launch cost minimization and payload maximization), constrained (ΔV, structural, and volume constraints), hierarchical (see Table 2.3) problem.

  In addition to the multi-objective version ("Rocket"), two single-objective versions are also available: 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 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.

- A Guidance, Navigation & Control (GNC) problem, adopted from [39, 122], which is a mixed-discrete (6 continuous and 11 discrete design variables), multi-objective (mass and failure rate minimization), hierarchical (see Table 2.3) problem. 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).

  Next to the multi-objective version ("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") are also implemented. The weight minimum lies in a group of one $\boldsymbol{x}$ of $x_{\text{valid,discr}}$ (1 / 327 = 0.3%).

- The jet engine architecture optimization problem of Section 2.6, which is a mixed-discrete (9 continuous and 6 discrete design variables), single-objective (fuel consumption minimization), constrained (5 inequality constraints), hierarchical (see Table 2.3) problem subject to hidden constraints. In a random DoE, the problem has a failure rate (FR) of 50%. The evaluation code of the test problem ("Jet SM") uses random forest regressors for each output to reduce evaluation times (milliseconds, compared to minutes for the original problem).

Table 2.3 presents some statistics for the problem versions used for testing optimization algorithms in this chapter: 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 imputation ratio IR, showing that they indeed feature hierarchical design spaces. The GNC and Jet SM problems need correction as they get about half their IR from the need for correction shown by a correction fraction CRF being a little over 50%. Max rate diversity (MRD) is shown as being common in SAO problems. The Jet SM problem contains hidden constraints, with an FR of 50%.

Table 2.3: Characteristics of test problems used for testing SAO algorithms. Abbreviations and symbols: n_f = number of objectives, n_g = number of inequality constraints, n_{x_c} = number of continuous design variables, n_{x_d} = number of discrete design variables, $n_{\text{valid,discr}}$ = number of valid discrete design vectors, IR = imputation ratio, CR = correction ratio, CRF = correction fraction, MRD = max rate diversity, FR = failure rate.

Problem	n_f	n_g	n_{x_c}	n_{x_d}	$n_{\text{valid,discr}}$	IR	CR	CRF	MRD	FR
Rocket	2	3	6	8	18 522	3.7	1.0	0%	94%	
MD GNC	2		6	11	327	150	16.1	56%	88%	
Jet SM	1	5	9	6	70	3.9	2.1	55%	60%	50%

2.3.2. Sampling Hierarchical Design Spaces

When sampling the design space to create the initial Design of Experiments (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 [39].

Sampling design vectors directly from the set of valid design vectors $x_{\text{valid,discr}}$ ensures generated design vectors are valid, and therefore correction is not needed after sampling: this is called the *hierarchical* sampling procedure. To prevent over- or under-representation, $x_{\text{valid,discr}}$ may be divided into subdivisions, and sampling can be separated in two steps: first decide how many samples to draw from each subdivision, then sample from the subdivisions. If $x_{\text{valid,discr}}$ is not available, a *non-hierarchical* sampling procedure must be used. Here, first discrete vectors are sampled randomly, then correction and imputation are applied to ensure that resulting design vectors are valid. Both the hierarchical and non-hierarchical samplers end by sampling continuous design variables by Sobol' sampling [225], ignoring inactive continuous design variables.

Defining Subdivisions Subdivisions (groups) can be defined based on design vector values and/or activeness information. In addition to not defining any subdivisions, the following ways to define subdivisions are considered in this investigation:

- Based on the number of active design variables n_{act} [226].
- Based on which design variables are active x_{act}.

  This approach is similar to how Frank et al. [130] 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 2.4 shows an example of how the valid discrete design vectors $x_{\text{valid,discr}}$ are separated into groups based on x_{act}.
- Based on values of high-RD (rate diversity) variables.

  Subdivisions are made by recursively selecting argmax RD_j subject to a minimum RD to ensure only high-RD variables are used for grouping.

Another way is to define subdivisions based on the value of certain discrete design variables. For example, Pelamatti et al. [132] define subdivisions based on dedicated dimensional variables, for example the selection of the number of components. 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). Therefore, this approach will not be considered further.

Table 2.4: Grouping and weighting process of the hierarchical sampling algorithm: discrete design vectors are grouped by x_{act} and weighted by $w = n_{\text{act}}$ (number of active x); w_{rel} represents the relative weighting for sampling design vectors from groups (e.g. $w_{\text{rel}} = 36\%$ 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}	$w = n_{\text{act}}$	w_{rel}
0	0	0	🟥	0	x_0, x_1 x_2, x_4	4	36%
0	0	0	🟥	1			
0	0	0	🟥	2			
0	0	1	0	🟥	x_0, x_1 x_2, x_3	4	36%
0	0	1	1	🟥			
0	0	2	0	🟥			
0	0	2	1	🟥			
0	1	🟥	🟥	🟥	x_0, x_1	2	18%
1	🟥	🟥	🟥	🟥	x_0	1	9%

Sampling from Subdivisions After subdivisions are defined, it should be determined how many samples to take from each subdivision by assigning weights w to each group. Three distributions are compared:

- uniform weighting as applied in [226]: $w = 1$;
- weighting by number of active design variables as applied in [132]: $w = n_{\text{act}}$; and
- weighting by group size: $w = \sqrt{n_{x,\text{rel}}}$, where $n_{x,\text{rel}}$ 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 group size uses $w = \sqrt{n_{x,\text{rel}}}$, because if $n_{x,\text{rel}}$ is used directly it is equivalent to hierarchical sampling without grouping. The square root is applied to prevent oversampling large groups.

Table 2.4 shows how the valid discrete design vectors $x_{\text{valid,discr}}$ 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.

Comparison of Sampling Algorithms (NSGA-II) First, the previously discussed sampling strategies are tested on NSGA-II. 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 (Eq. (E.3)) using the procedure described in Appendix E. ΔHV (Δ hypervolume; (Eq. (E.1))) represents the distance to the known optimum (or Pareto front in case of multi-objective optimization) normalized to the range of objective values. A lower ΔHV regret is better, as it shows that the optimum was approached more closely and/or reached sooner. Table 2.5 presents the results of sampling strategies for NSGA-II. The columns before the "rank" columns present the strategy performance ranks per test problem: the best performing strategies are assigned rank 1; higher ranks are progressively assigned to worse performing strategies. Higher percentages in the rank columns represent a strategy more often reaching the associated rank, as seen over all test problems. The penalty represents an increase of mean ΔHV regret compared to the best strategy: lower is better. Throughout the table, darker background colors represent better results. The overall best performing strategy is underlined. For more details on how to interpret such results tables, refer to Appendix E.1.

Table 2.5 shows that 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 represented by only 0.3% of $x_{\text{valid,discr}}$ (1 sensor/computer for the GNC problem; 1 stage for the Rocket problem). The non-grouping sampler uniformly samples over all $x_{\text{valid,discr}}$ 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.

Table 2.5: 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. Appendix E.1 explains how to interpret these kinds of results in more details. 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 2.6: 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. Appendix E.1 explains how to interpret these kinds of results in more details. 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%

2

Comparison of Sampling Algorithms (BO) For the BO algorithm, the non-hierarchical sampler and hierarchical non-weighted samplers are compared. The BO algorithm is executed with $n_{\text{doe}} = 3 \cdot n_x$, 40 infill points and 24 repetitions. For the Jet SM problem, $n_{\text{doe}} = 10 \cdot n_x$ for BO will be used, to correct for the fact that this problem features a hidden constraint and therefore needs a larger DoE. Table 2.6 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.

Non-hierarchical sampling performs slightly worse than the best sampler, at a 12% mean performance penalty both for NSGA-II and for the BO algorithm. Therefore, although the availability of $x_{\text{valid,discr}}$ improves algorithm performance, the non-availability does not prevent the problem from being solved.

When comparing between NSGA-II and the BO algorithm, therefore, x_{act} and MRD are good candidates for hierarchical sampling. Hierarchical sampling based on x_{act} is selected for its slightly better performance on the BO algorithm.

2.3.3. Design Vector Correction Algorithms

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 a high correction ratio (CR) due to the low chance of randomly finding a correct design vector. This investigation only considers correction of discrete design variables, as in this work it is assumed that only discrete variable determine hierarchy.

The mechanism of correction depends on the hierarchical structure of the optimization problem and therefore can be implemented on a per-problem basis, known as *problem-specific* correction. In such a case, 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 correct 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 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 may reduce implementation effort and potentially improve optimizer performance. Two classes of correction algorithms are defined: eager algorithms for when $x_{\text{valid,discr}}$ is available, and lazy algorithms for when it is not.

Eager Problem-Agnostic Correction The following *eager* correction algorithms have been developed as part of this work:

- *Any-select*: As eager correction algorithms have access to $x_{\text{valid,discr}}$, the simplest algorithm selects any of the available vectors as a replacement. A variant that always returns the first (or any index for that matter) valid design vector can be defined, however that would not help with exploration at all: for problems with high correction 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 2.3b.

- *Greedy*: this algorithm (see Figure 2.3c) repeatedly filters $x_{\text{valid,discr}}$ based on the selected value of each design variable, starting from the left of the design vector to be corrected x_{corr}. If a given design variable value leaves no valid design vectors to be selected from, it is corrected to the closest value that does.
- *Similarity*: eager correction based on similarity (see Figure 2.3d) is done by calculating the weighted distances from x_{corr} to all vectors in $x_{\text{valid,discr}}$ 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 to favor changes on the right side of the design vector over changes on the left, assuming that left-side design variables represent higher-impact choices. As distance metrics either Euclidean (d_{euc}) or Manhattan (d_{manh}) distance can be used:

$$d_{\text{euc}}(\boldsymbol{x}, \boldsymbol{x}') = \sqrt{\sum_{i=1}^{n_x} \left(x_i - x_i'\right)^2}, \tag{2.11}$$

$$d_{\text{manh}}(\boldsymbol{x}, \boldsymbol{x}') = \sum_{i=1}^{n_x} \left|x_i - x_i'\right|. \tag{2.12}$$

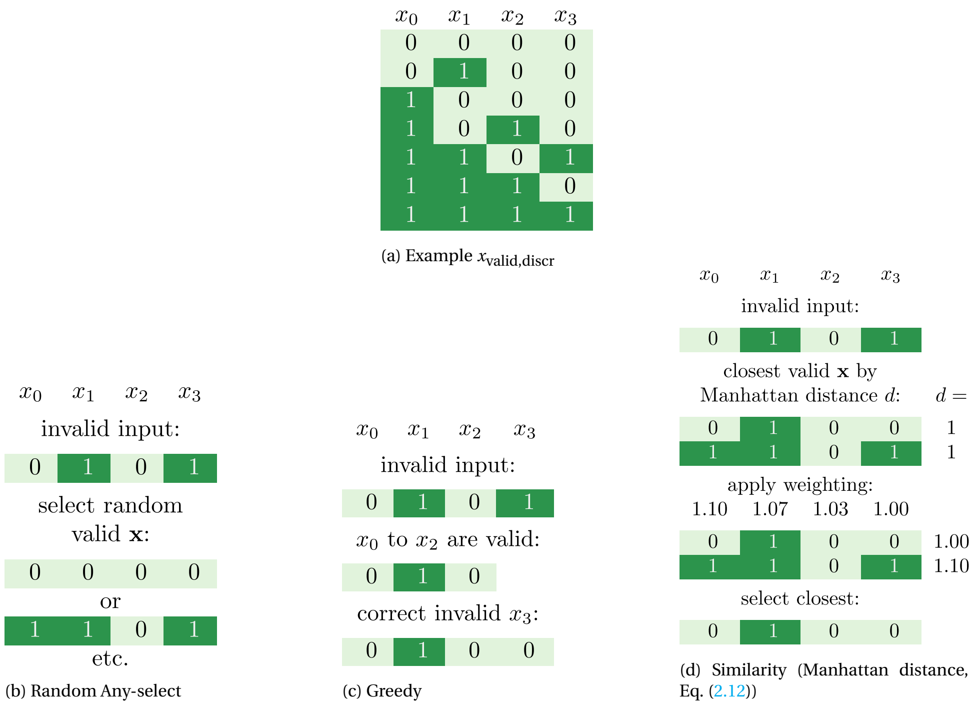


Figure 2.3: A visualization of different eager correction strategies.

Lazy Problem-Agnostic Correction *Lazy* correction algorithms do not have access to $x_{\text{valid,discr}}$ and instead provide some way to generate design vectors and then check with a user-provided function isCorrect($\boldsymbol{x}$) whether that design vector is correct or not. This generation and checking process continues until a correct design vector is found. The following lazy correction algorithms have been developed:

- *Any-select*: analogous to the any-selection eager algorithm, also an any-selection lazy algorithm can be defined. Randomized selection is used, which on average needs to generate CR (correction ratio) design vectors before finding a correct design vector. This is the main limitation of lazy compared to eager correction algorithms.
- *Similarity*: lazy correction by similarity is done by modifying x_{corr} with some Δ_{corr} vector, which can either be generated depth-first or distance-first:
  - Depth-first directly applies the generated Cartesian product of Δ values for each design variable.
  - Distance-first 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.

Note that greedy correction is not possible for lazy algorithms, because it is assumed that isCorrect($\boldsymbol{x}$) only operates on complete design vectors and not fractions of design vectors as would be needed for greedy correction.

Table 2.7 presents an overview of discussed correction strategies.

Table 2.7: Overview of correction algorithms for hierarchical design spaces. Euc and Manh refer to Euclidean and Manhattan distance metrics, respectively. See Figure 2.3 for a visualization of eager correction algorithms.

Type	Algorithm	Configuration	Requires
Problem-specific			Custom correction function
Eager	Any-select		$x_{\text{valid,discr}}$
	Greedy		$x_{\text{valid,discr}}$
	Similarity	Euclidean distance	$x_{\text{valid,discr}}$
		Manhattan distance	$x_{\text{valid,discr}}$
Lazy	Any-select		isCorrect($\boldsymbol{x}$)
	Similarity	Depth-first	isCorrect($\boldsymbol{x}$)
		Distance-first; Euclidean distance	isCorrect($\boldsymbol{x}$)
		Distance-first; Manhattan distance	isCorrect($\boldsymbol{x}$)

Comparison of Correction Algorithms (NSGA-II) 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 $x_{\text{valid,discr}}$ is not available. Correction strategies are tested for the various algorithm-specific configuration options (see Table 2.7). Eager and lazy correction are additionally compared to problem-specific correction.

Table 2.8 presents results of correction strategies for NSGA-II. Eager correction performs better than lazy correction, and comparable to problem-specific correction. Eager Any-select performs best, with Eager Similarity (Manhattan or Euclidean distance) performing comparably. Lazy Similarity (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 correction ratio CR, because it is based on a trial-and-error approach.

Comparison of Correction Algorithms (BO) For the BO algorithm, best performing eager and lazy correction algorithms and problem-specific correction are compared. The BO algorithm is executed with $n_{\text{doe}} = 3 \cdot n_x$ ($n_{\text{doe}} = 10 \cdot n_x$ for the Jet SM problem), 40 infill points and 24 repetitions. Table 2.9 presents sampling results for the BO algorithm. Problem-specific correction with hierarchical sampling performs best. Eager correction performs comparable 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, it is concluded that problem-specific greedy 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 $x_{\text{valid,discr}}$ is available, then Eager Any-select correction should be used.

Table 2.8: 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. Appendix E.1 explains how to interpret these kinds of results in more details.

Correction	Config	Sampling	RCost	RWt	Rocket	SO GNC	FR GNC	[1]	[2]	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 Similarity	Manh	Hier. x_{act}	1	1	1	1	3	2	1	2	62%	88%	3%
Eager Similarity	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 Similarity	Manh; Dist-f.	Non-hier.	3	1	3	3	4	3	1	3	25%	25%	15%
Lazy Similarity	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%

[1] Wt GNC
[2] MD GNC

Table 2.9: 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. Appendix E.1 explains how to interpret these kinds of results in more details.

Correction	Config	Sampling	RCost	RWt	Rocket	SO GNC	[1]	[2]	Jet SM	Rank 1	Rank ≤ 2	Penalty
Eager Any-select		Hier. x_{act}	1	1	1	2	1	2	1	71%	100%	2%
Eager Similarity	Manh	Hier. x_{act}	1	1	2	2	1	1	1	71%	100%	3%
Lazy Similarity	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%

[1] FR GNC
[2] MD GNC

2.3.4. USING HIERARCHY INFORMATION IN OPTIMIZATION ALGORITHMS

The hierarchical sampling and correction algorithms investigated in the preceding sections assume that all information about the hierarchical design space is readily available: design vectors are corrected and imputed, information about design variable activeness is available, and optionally the set of all valid discrete design vectors $x_{\text{valid,discr}}$ is available. This section discusses how such information can be integrated in existing optimization algorithms, and investigates whether it indeed improves optimization algorithm performance. According to ZAEFFERER ET AL. [133] there are three high-level strategies for integrating information about 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*: correction and imputation are applied to ensure that all evaluated design vectors are valid.
3. *Explicit consideration*: the hierarchical structure is explicitly made available to and used by the optimization algorithm.

Figure 2.4 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. A discussion of the three strategies follows.

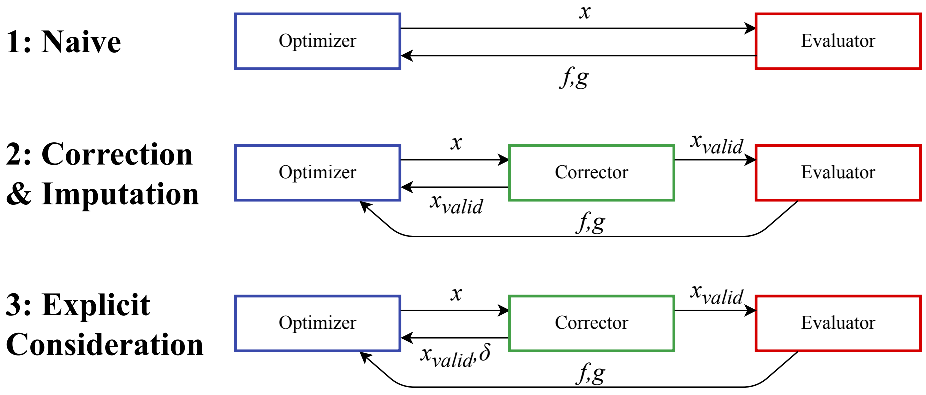


Figure 2.4: High-level strategies for dealing with hierarchical optimization.

Naive With the naive approach, the evaluation function is left unmodified compared to non-hierarchical optimization:

$$f, g = \text{evaluate}(\boldsymbol{x}). \tag{2.13}$$

The results of this is that there might be a discrepancy between which design vectors the optimizer thinks are being evaluated and which design vectors actually are being evaluated. 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. For example, the optimizer might decide to generate new design vectors by modifying inactive variables of a previously-evaluated design vector, thereby effectively exploring the same already-explored design vector. This effect is especially present for problems with a high imputation ratio IR, because of the low chance of randomly finding new valid design vectors in that case.

Correction and Imputation As discussed in Section 1.5.3, *correction and imputation* ensure that design vectors are correct by ensuring value constraints (constraints that restrict options of design variables based on selected options of other design variables) are satisfied and by imputing inactive design variables to some default value, respectively. The result is a valid design vector $\boldsymbol{x}_{\text{valid}}$ that then is evaluated. Applying these operations avoids the design vector mapping problems encountered in the "naive" strategy. Applying correction and imputation requires that the optimization algorithm supports the fact that the design vector might be modified by the evaluation function, and that it records such modifications in its database of evaluated design vectors. With the correction and imputation approach, Eq. (2.13) is modified to:

$$f, g, \boldsymbol{x}_{\text{valid}} = \text{evaluate}\left(\boldsymbol{x}\right). \tag{2.14}$$

Another way to support correction and imputation is through an *ask-and-tell* interface [227]: 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.

Correction and imputation can also be implemented using a *repair* operator [228]: a problem-specific function that modifies design vectors. The advantage of a repair operator over 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. The standalone repair operator can be formulated as:

$$\boldsymbol{x}_{\text{valid}} = \text{repair}\left(\boldsymbol{x}\right). \tag{2.15}$$

This allows correction and imputation to be applied during other steps in the optimization process than evaluation, for example when generating initial design points or when searching the design space for the best infill point for surrogate-based optimization algorithms [175].

Explicit Consideration 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 is made available to the optimizer. Activeness information comprises both the activeness vector δ for a given design vector x, and the list of which design variables are conditionally active. The availability of activeness information makes it possible to generate all possible valid design vectors $x_{\text{valid,discr}}$, and therefore it allows using the hierarchical sampling algorithm presented in Section 2.3.2. Also, it enables applying hierarchical kernels in GP models used by BO [26, 134, 140, 182]. When using the explicit consideration strategy, the evaluation function (Eq. (2.14)) is modified to:

$$f, g, \boldsymbol{x}_{\text{valid}}, \delta = \text{evaluate}\left(\boldsymbol{x}\right). \tag{2.16}$$

In addition, the repair operator (Eq. (2.15)) is modified to:

$$\boldsymbol{x}_{\text{valid}}, \delta = \text{repair}\left(\boldsymbol{x}\right). \tag{2.17}$$

Finally, the problem also returns $x_{\text{valid,discr}}$, $\delta_{\text{valid,discr}}$, and information about which design variables are conditionally active. Note that returning all valid design vectors $x_{\text{valid,discr}}$ and associated activeness information is optional, because it might not be possible to determine them all due to time or memory limits. Adherence to these interfaces has been implemented by the author in the Surrogate Modeling Toolbox (SMT) [26] as part of this work.

Modeling the Hierarchical Design Space One way to explicitly consider the hierarchical structure is by formally modeling the hierarchical structure and making this model available to the optimization algorithm. Because the model then 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. [132, 181].
2. Tree-structured: conditional variables can also determine activeness of other variables, e.g. BoTorch [229].
3. Directed acyclic graph: activeness can be determined by multiple variables, e.g. [134, 188] and ConfigSpace [230].
4. Directed graph: additionally supports cyclic dependencies.

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. Table 2.10 presents a detailed overview of the discussed integration strategies. It breaks down the three high-level strategies into various levels of additional capabilities gained, based on the data interfaces being made available to the optimizer, and based on whether the design space model is made available explicitly.

Table 2.10: 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 $x_{\text{valid,discr}}$.
[4] Hierarchical kernels for surrogate modeling.
[5] Dedicated search operators and possibility for problem decomposition.
[6] Physical separation between optimization and evaluation code.

Comparison of Strategies The different levels of integration are now compared to investigate if applying higher levels of integration indeed results in better optimizer performance. Experiments are run for the Naive, x-output, Repair and Activeness integration strategies. For BO, the activeness strategy is run in two configurations: one where activeness is only used for hierarchical sampling, and one where activeness is available additionally for the GP models. An overview of tested integration strategies is provided in Table 2.11. NSGA-II is executed with $n_{\text{doe}} = 10 \cdot n_x$, 25 generations and 100 repetitions. The BO algorithm is executed with $n_{\text{doe}} = 3 \cdot n_x$ ($n_{\text{doe}} = 10 \cdot n_x$ for the Jet SM problem), 40 infill points and 16 repetitions.

Table 2.11: Tested hierarchy integration strategies and impact on available capabilities. Abbreviations: hier. = hierarchical, corr. = correction, expl. cons. = explicit consideration.

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

Table 2.12 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 2.13 shows that for BO a higher level of integration improves optimizer performance. Naive, x-output 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. However, a reduction in performance when using hierarchical kernels (Activeness) compared to non-hierarchical kernels (Hierarchical sampling) is observed. Hierarchical kernels work well for querying hierarchical models [26], however might work less well when subject to clustering of sampling points as would occur in an SBO run.

Table 2.12: 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. Appendix E.1 explains how to interpret these kinds of results in more details.

	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-output	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 2.13: 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. Appendix E.1 explains how to interpret these kinds of results in more details.

	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-output	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%

2.4. Bayesian Optimization with Hidden Constraints

This section identifies and investigates various strategies for dealing with hidden constraints in Bayesian Optimization (BO) algorithms.

Rejection Strategy The simplest strategy is to train the surrogate models only on viable points, thereby in effect *rejecting* failed points from the training set [147]. The disadvantage to this approach is that knowledge of the design space is ignored: the optimizer might get stuck suggesting the same infill point(s) over and over, because it cannot know that these infill points will fail to evaluate.

Replacement Strategy A more advanced approach is to *replace* the failed points by some values derived from viable points, as initially suggested by Forrester et al. [146] and inspired by imputation in the sense of replacing missing data in statistical datasets. They reason that failed points actually represent missing data and can be replaced by values of close-by viable points. However, the replaced values should drive the optimizer towards the viable region of the design space, which leads them to formulate a method for finding replacement values from a Gaussian Process (GP) model trained on the viable points only:

$$y_{\text{replace}}\left(\boldsymbol{x}_{\text{failed}}\right) = \hat{y}\left(\boldsymbol{x}_{\text{failed}}\right) + \alpha\, \hat{\sigma}\left(\boldsymbol{x}_{\text{failed}}\right), \tag{2.18}$$

where y represents the output value to be replaced, $\hat{y}$ (Eq. (A.4)) and $\hat{\sigma}$ (Eq. (A.5)) the output and uncertainty estimates of the GP trained on viable design points, and α some multiplier ($\alpha = 1$ in [146]). They show that their approach works well for a continuous single-objective airfoil optimization problem.

Another strategy for replacing values of failed points is by simply selecting one or more neighbor points and applying some aggregation function to get one value to replace. For example, Huyer & Neumaier [231] replace failed values by the max of n_{nb} neighbor points. This concept can be expanded by considering $n_{nb} = 1$ to use the closest point to select the replacement value, or $n_{\text{nb}} = n_{\text{viable}}$ to consider all viable points for the replacement value.

Prediction Strategy The most advanced method for dealing with hidden constraints in SBO algorithms is to *predict* where the failed region lies with the help of another model [232]. It is possible to do this based on distances, such as hyperspheres [233] or Voronoi tesselation [234], however these only work for continuous design spaces.

For mixed-discrete and hierarchical design spaces the application of machine learning models is investigated. Here the idea is as follows:

1. Assign binary labels to all design points based on their failure status: 0 for failed points and 1 for viable points.
2. Train a model on these labels.
3. Use the model to predict the Probability of Viability (PoV):

$$\text{PoV}(\boldsymbol{x}) = \hat{y}_{\text{PoV}}(\boldsymbol{x}), \tag{2.19}$$

where $\hat{y}_{\text{PoV}}$ is the value predicted by the model.

PoV lies between 0 (0%) and 1 (100%), and therefore represents the predicted probability that a newly-selected infill point will be viable (i.e. will not fail). During infill optimization, PoV can be used in two ways: either as a penalty multiplier to the infill criterion [232], or as a constraint to the infill problem that ensures that the PoV of selected infill points is at or above some threshold [235].

Different types of surrogate models have been used to predict PoV, mainly selected due to their ability for modeling such binary classification problems. Used models include Random Forest Classifiers (RFC) [232], piecewise linear Radial Basis Functions (RBF) [147], Support Vector Machines (SVM) [236], SVM's with RBF kernel [235], Gaussian Process models [237–239], and K-Nearest Neighbors (KNN) classifiers [239, 240].

Implementation Approaches The rejection and replacement strategies are implemented into the BO algorithm in a preprocessing step before training the GP models. The training set is separated into a viable and a failed set; for rejection the viable set is then simply discarded, whereas for replacement the points in the failed set are assigned some value for each output that is based on the viable points. Afterwards, the GP models are trained and the infill selection process continues as usual.

The integration of the prediction strategy requires the modification of the infill process itself. As for rejection and replacement, the training set is separated into the viable and failed set. The viable set is then directly used to train the GP models for infill search. The additional surrogate model for Probability of Viability (PoV) prediction is trained with a set containing all points and with binary labels assigned according to the viability status of the points: 0 for failed points and 1 for viable points. The infill optimization problem can then be modified by adding an inequality constraint that ensures that $\mathrm{PoV}(\boldsymbol{x}) \geq \mathrm{PoV}_{\mathrm{min}}$:

$$g_{\mathrm{PoV}}(\boldsymbol{x}) = \mathrm{PoV}_{\mathrm{min}} - \mathrm{PoV}(\boldsymbol{x}) \leq 0, \tag{2.20}$$

where $\mathrm{PoV}_{\mathrm{min}}$ represents a user-defined minimum PoV to be reached, and $\mathrm{PoV}(\boldsymbol{x})$ (Eq. (2.19)) represents the predicted PoV for a given design point. PoV can also be integrated by modifying the infill objectives:

$$f_{\mathrm{m,infill,mod}}(\boldsymbol{x}) = 1 - \left(\left(1 - f_{\mathrm{m,infill}}(\boldsymbol{x})\right)\, \mathrm{PoV}(\boldsymbol{x})\right), \tag{2.21}$$

where $f_{\mathrm{m,infill}}$ represents the m^{th} infill objective, and assuming infill objectives are normalized and to be minimized.

To illustrate this process, Figure 2.5 presents several steps of running BO on a test problem, with an RBF model as PoV predictor used as an infill constraint (Eq. (2.20)) with $\mathrm{PoV}_{\mathrm{min}} = 50\%$. The test problem is the single-objective problem from ALIMO ET AL. [235], modified to have its optimum at the edge of the failed region near the bottom of the 2D design space. As can be seen, all suggested infill points satisfy the g_{PoV} constraint, however, in the earlier iterations this constraint can be inaccurate. Several infills are generated that violate the hidden constraint, and after each iteration the model gets more accurate at the edges of the failed regions for the three locations in the design space where the optimizer expects the optimum to lie. Additionally, it is shown that the model should be able to handle closely-spaced failed and viable points.

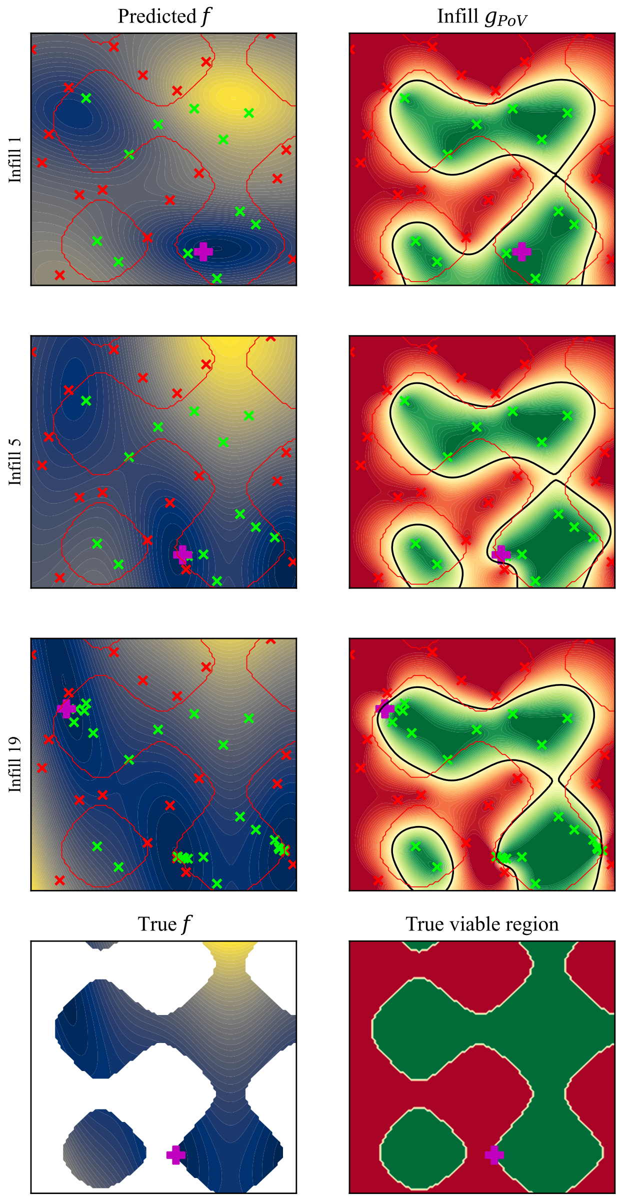


Figure 2.5: Several optimization steps between iteration 1 and 20 of BO executed on a test problem with its optimum lying at the edge of the failed region, as shown in the bottom row. The main GP is shown on the left (darker means a lower, more optimal value), and the RBF model for predicting PoV is shown on the right. The RBF model is used as an infill constraint with $\text{PoV}_{\text{min}} = 50\%$, showing green and red contours for satisfied and violated constraint values, respectively. Green, red, and magenta points represent viable, failed, and selected infill points, respectively.

Comparison of Strategies To compare strategy performance, the test problems listed in Table 2.14 are used to run the set of hidden constraint strategies listed in Table 2.15. Compared to the previously identified predictor models, additionally a Variational GP and the mixed-discrete GP developed in [241] are tested. A Variational GP does not assume a Gaussian distribution and therefore might be able to more accurately model discontinuous functions as seen in classification problems [180, 242]. To ensure there are enough viable points to train models on for the infill search, the DoE size of problems containing hidden constraints should be increased:

$$n_{\text{doe}} = \frac{k_{\text{doe}}\, n_x}{1 - \text{FR}_{\text{exp}}}, \tag{2.22}$$

where k_{doe} is the DoE multiplier, n_x the number of design variables, and FR_{exp} the expected fail rate. An expected fail rate of 60% and $k_{\text{doe}} = 2$ (k_{doe} can be relatively small for BO, compared to creating a DoE for constructing a surrogate model that is accurate throughout the design space) are used for the following tests. Each optimization is executed with $n_{\text{infill}} = 50$ and is repeated 8 times.

Table 2.14: Test problems for comparing hidden constraint strategies. Abbreviations and symbols: IR = imputation ratio, FR = failure rate, HC = hidden constraints, MD = mixed-discrete, H = hierarchical, MO = multi-objective, n_{x_c} and n_{x_d} = number of continuous and discrete design variables, respectively, n_f = number of objectives, n_g = number of constraints.

Name	Ref.	n_{x_c}	n_{x_d}	n_f	n_g	IR	FR
Branin	[166]	2		1			0%
HC Branin	[237]	2		1			33%
Alimo	[235]	2		1			51%
Alimo Edge		2		1			53%
HC Sphere	[236]	2		1			51%
Müller 1	[147]	5		1			67%
Müller 2	[147]	4		1			40%
HC CantBeam		4		1	1		83%
HC Carside Less		7		1	9		39%
HC Carside		7		3	8		66%
MD/HC CantBeam		2	2	1	1		81%
MD/HC Carside		3	4	3	8		66%
H Alimo		2	5	1		5.4	51%
H Alimo Edge		2	5	1		5.4	53%
H Müller 2		4	4	1		5.4	37%
H/HC Rosenbrock	[243]	8	5	1	1	1.5	21%
MO/H/HC Rosenbrock	[243]	8	5	2	1	1.5	60%
Jet SM	Section 2.3.1	9	6	1	5	3.9	50%

Table 2.15: Tested Bayesian Optimization hidden constraint strategies. Abbreviations: GP = Gaussian Process.

Strategy	Sub-strategy	Configuration	Implementation
Rejection			
Replacement	Neighborhood	Global, max	
		Local	
		5-nearest, max	
		5-nearest, mean	
	Predicted worst	$\alpha = 1$	SMT[1]
		$\alpha = 2$	SMT[1]
Prediction	Random Forest Classifier	$\text{PoV}_{\text{min}} = 50\%$	scikit-learn[2]
	K-Nearest Neighbors	$\text{PoV}_{\text{min}} = 50\%$	scikit-learn[2]
	Radial Basis Function	$\text{PoV}_{\text{min}} = 50\%$	Scipy[3]
	GP Classifier	$\text{PoV}_{\text{min}} = 50\%$	scikit-learn[2]
	Variational GP	$\text{PoV}_{\text{min}} = 50\%$	Trieste[4]
	Mixed-Discrete GP	$\text{PoV}_{\text{min}} = 50\%$	SMT[1]

[1] https://smt.readthedocs.io/
[2] https://scikit-learn.org/
[3] https://scipy.org/
[4] https://secondmind-labs.github.io/trieste/

Table 2.16 presents optimization performance results. It can be seen that prediction with a Random Forest Classifier (RFC) performs best (rank 1) or good (rank $\leq$ 2) compared to other strategies, with mixed-discrete (MD) GP prediction and predicted worst replacement ($\alpha = 1$) closely following. Table 2.17 presents performance measures averaged over all test problems, relative to the rejection strategy. A reduction in ΔHV regret is usually combined with a reduction in failure rate, as a lower failure rate indicates a better capability of avoiding the failed region. Replacement strategies approximately double the training and infill cycle time, due to the fact that the trained GP models (one for each f and g) contain the complete set of design points as training values, whereas for rejection and prediction only the viable points are used. Prediction strategies train an extra model to predict the PoV, which leads to a moderate increase in training and infill time.

Table 2.16: Comparison of hidden constraint strategies on various test problems, ranked by ΔHV regret (lower rank / darker color is better). Best performing strategy is underlined. Appendix E.1 explains how to interpret these kinds of results in more details. Abbreviations: HC = hidden constraints, MD = mixed-discrete, H = hierarchical, MO = multi-objective, RFC = Random Forest Classifier, KNN = K-Nearest Neighbors, RBF = Radial Basis Functions, GP = Gaussian Process.

		Branin	HC Branin	Alimo	Alimo Edge	Müller 2	HC Sphere	Müller 1	HC CantB	[1]	HC Carside
Rejection		1	4	2	4	1	2	2	5	6	6
Replacement	Global max	2	3	2	5	1	4	3	2	3	4
Replacement	Local	2	2	1	4	3	2	2	4	5	5
Replacement	5-nearest, max	2	2	1	3	1	2	1	3	4	3
Replacement	5-nearest, mean	1	2	1	3	2	1	1	4	2	2
Replacement	Predicted worst	2	1	1	4	1	3	1	1	2	2
Replacement	Pred. worst ($\alpha = 2$)	2	2	2	3	2	4	2	1	3	3
Prediction	RFC	1	2	1	1	1	2	1	1	3	1
Prediction	KNN	2	2	1	3	2	2	2	2	3	4
Prediction	RBF	2	1	1	2	1	3	3	1	2	2
Prediction	GP Classifier	1	1	1	1	1	2	2	1	3	3
Prediction	Variational GP	1	1	1	1	1	3	4	1	4	2
Prediction	MD GP	2	1	1	2	1	2	1	2	1	1

		MD/HC CantB	MD/HC Carside	H Alimo	H Alimo Edge	H Müller 2	H/HC Rosenbr.	MO/H/HC Rbr.	Jet SM	Rank 1	Rank ≤ 2
Rejection		5	3	4	4	2	3	3	4	11%	33%
Replacement	Global max	2	2	2	3	1	4	5	3	11%	44%
Replacement	Local	4	3	3	3	2	2	4	3	6%	39%
Replacement	5-nearest, max	3	2	2	2	1	2	4	2	22%	67%
Replacement	5-nearest, mean	3	1	1	2	1	2	2	2	39%	83%
Replacement	Predicted worst	2	1	1	2	1	1	3	1	56%	83%
Replacement	Pred. worst ($\alpha = 2$)	1	1	1	1	1	2	3	1	39%	72%
Prediction	RFC	1	1	1	2	1	1	1	1	78%	94%
Prediction	KNN	2	2	3	4	1	3	1	2	17%	67%
Prediction	RBF	2	1	2	3	1	1	1	1	50%	83%
Prediction	GP Classifier	1	1	3	3	1	2	1	1	61%	78%
Prediction	Variational GP	1	1	2	3	1	2	2	1	56%	78%
Prediction	MD GP	1	1	1	1	1	1	5	1	72%	94%

[1] HC Carside Less

Table 2.17: Performance of hidden constraint strategies relative to the rejection strategy, averaged over all test problems at the end of the optimization runs. Darker color is better for ΔHV regret and Fail rate; worse for time columns. Abbreviations: HV = hypervolume, RFC = Random Forest Classifier, KNN = K-Nearest Neighbors, RBF = Radial Basis Functions, GP = Gaussian Process.

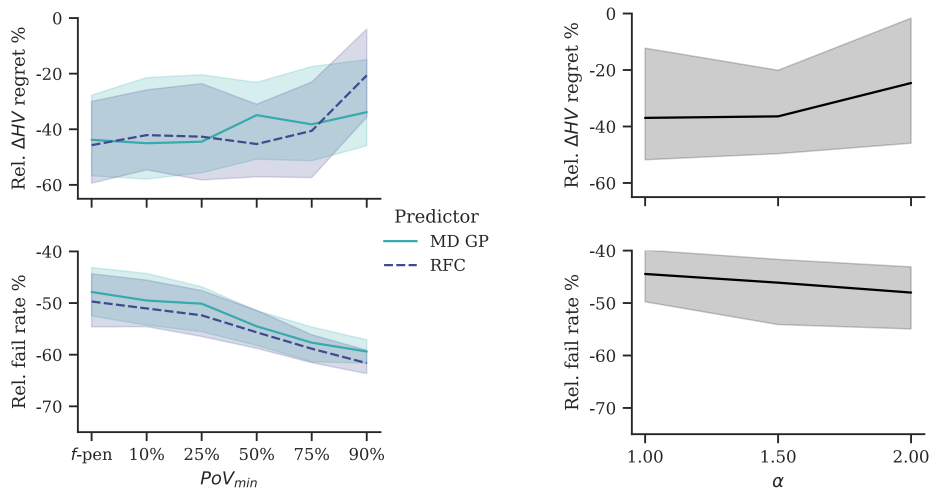
Strategy	Sub-strategy	ΔHV regret	Fail rate	Training time	Infill time	Training + infill time
Rejection		+0%	+0%	+0%	+0%	+0%
Replacement	Global max	-9%	-61%	+199%	+74%	+74%
Replacement	Local	-13%	-15%	+202%	+76%	+79%
Replacement	5-nearest, max	-30%	-43%	+207%	+78%	+84%
Replacement	5-nearest, mean	-35%	-23%	+193%	+78%	+77%
Replacement	Predicted worst	-33%	-48%	+192%	+67%	+73%
Replacement	Pred. worst ($\alpha = 2$)	-30%	-53%	+199%	+75%	+78%
Prediction	RFC	-44%	-61%	+86%	+94%	+83%
Prediction	KNN	-27%	-39%	+48%	+59%	+49%
Prediction	RBF	-35%	-61%	+126%	+87%	+80%
Prediction	GP Classifier	-37%	-58%	+106%	+156%	+114%
Prediction	Variational GP	-32%	-60%	+68%	+243%	+189%
Prediction	MD GP	-38%	-62%	+116%	+95%	+91%

Parameter Studies Predicted worst replacement, prediction with RFC, and prediction with MD GP are selected as most promising candidates. These three strategies are further investigated to find out the influence of their parameter settings: α for predicted worst replacement, and PoV_{min} for the prediction strategies. In addition, the prediction strategies are tested with integration as f-infill penalty (Eq. (2.21)). Tests are run with the same settings as the previous experiment.

Figure 2.6 shows the relative improvement over the rejection strategy for the tested strategy configurations. Fail rate is reduced significantly for higher values of PoV_{min} and α, which is expected as higher values result in a more conservative approach and therefore less exploration of the failed region. A reduction in failure rate, however, decreases optimizer performance (seen by an increase in ΔHV regret), showing that a certain amount of failed evaluations is needed to sufficiently explore the design space. This is needed both for exploring new regions of the design space where the data about the hidden constraint is inaccurate, and for ensuring that the border of the hidden constraint is accurate if the optimum of the problem lies near the constraint boundary.

Table 2.18 presents ranking of algorithm performance. The best performing strategies are the prediction strategies at low PoV_{min} values. It also shows that f-infill penalty behaves similar as low PoV_{min} values. Prediction with MD GP or RFC are behaving similarly-well, although MD GP for a little wider range of PoV_{min} than RFC.

From these results, it is concluded that either MD GP or RFC prediction should be used to deal with hidden constraints in BO. PoV should be integrated as a constraint, because it allows more control over exploration vs exploitation compared to integration as f-infill penalty. PoV_{min} should be kept relatively low to promote sufficient exploration: a value of $\text{PoV}_{\text{min}} = 25\%$ will be used subsequently.



(a) Predictor strategies

(b) Predicted worst replacement strategy

Figure 2.6: Comparison of hidden constraint strategy settings relative to the rejection strategy, averaged over all test problems at the end of the optimization runs (repeated 8 times). Abbreviations: HV = hypervolume, MD = mixed-discrete, GP = Gaussian Process, RFC = Random Forest Classifier.

Table 2.18: Comparison of hidden constraint strategy settings on various test problems, ranked by ΔHV regret (lower rank / darker color is better). Best performing strategy is underlined. Appendix E.1 explains how to interpret these kinds of results in more details. Abbreviations: HC = hidden constraints, MD = mixed-discrete, H = hierarchical, MO = multi-objective, RFC = Random Forest Classifier, KNN = K-Nearest Neighbors, RBF = Radial Basis Functions, GP = Gaussian Process.

		Alimo	Alimo Edge	Müller 1	[1]	H Alimo	H/HC Rbr.	[2]	Jet SM	Rank 1	Rank ≤ 2	Penalty
Predicted Worst	$\alpha = 1.00$	1	3	2	3	1	2	2	2	25%	75%	15%
Predicted Worst	$\alpha = 1.50$	1	3	2	2	1	1	2	2	38%	88%	13%
Predicted Worst	$\alpha = 2.00$	2	4	1	3	2	2	3	2	12%	62%	36%
<u>MD-GP</u>	<u>f_{infill} penalty</u>	1	1	1	1	2	2	1	1	<u>75%</u>	<u>100%</u>	<u>1%</u>
MD-GP	$\text{PoV}_{\text{min}} = 10\%$	1	2	1	1	2	2	1	1	62%	100%	3%
<u>MD-GP</u>	<u>$\text{PoV}_{\text{min}} = 25\%$</u>	1	2	1	1	1	2	1	1	<u>75%</u>	<u>100%</u>	<u>-1%</u>
MD-GP	$\text{PoV}_{\text{min}} = 50\%$	1	2	1	1	1	2	4	1	62%	88%	9%
MD-GP	$\text{PoV}_{\text{min}} = 75\%$	1	2	2	1	1	2	3	1	50%	88%	7%
MD-GP	$\text{PoV}_{\text{min}} = 90\%$	1	2	2	1	1	2	4	1	50%	88%	13%
RFC	f_{infill} penalty	1	1	2	3	2	1	1	1	62%	88%	-2%
RFC	$\text{PoV}_{\text{min}} = 10\%$	1	1	3	2	2	2	1	2	38%	88%	10%
RFC	$\text{PoV}_{\text{min}} = 25\%$	1	1	2	2	2	2	1	1	50%	100%	6%
RFC	$\text{PoV}_{\text{min}} = 50\%$	1	2	2	3	2	1	1	1	50%	88%	1%
RFC	$\text{PoV}_{\text{min}} = 75\%$	1	2	2	1	2	1	2	2	38%	100%	9%
RFC	$\text{PoV}_{\text{min}} = 90\%$	1	3	4	4	2	2	4	3	12%	38%	54%

[1] MD/HC Carside
[2] MO/H/HC Rosenbrock

2.5. IMPLEMENTATION IN SBARCHOPT

As part of this work, the results of this chapter are implemented in SBArchOpt (Surrogate-Based Architecture Optimization)[2]: an open-source Python library for solving SAO problems.

SBArchOpt features a *problem definition* class with interfaces for executing hierarchical optimization problems, developed to support the highest level of hierarchy integration as discussed in Section 2.3.4. The problem definition class is built on top of pymoo's[3] PROBLEM class [244], extends the evaluation function according to Eq. (2.16), adds functions for performing imputation and correction according to Eq. (2.17), and makes these available as a standalone repair operator. Additional functions are provided for getting information about which design variables are conditionally active, for generating all valid discrete design vectors $x_{\text{valid,discr}}$, and for calculating various statistics such as IR, CR, CRF, and MRD (see Section 2.2). 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] [230].

Next to the problem definition class, SBArchOpt also implements the following *optimization algorithms*:

- DOEALGORITHM: an optimization algorithm that only runs the hierarchical sampling algorithm presented in Section 2.3.2.

  The sampling algorithm itself is implemented as HIERARCHICALSAMPLING, a pymoo SAMPLER class.
- ARCHOPTNSGA2: NSGA-II that uses the hierarchical sampling algorithm, and supports results storage and restart.
- ARCHSBO: the BO algorithm developed in this chapter, including the hierarchical sampling algorithm, support for results storage and restart, the possibility to either use an RBF or a GP model (both using the implementation in SMT [26]), and automatic selection of ensemble infill and constraint handling strategies (see Section 2.1) and the hidden constraint strategy (see Section 2.4).
- To test and promote solving SAO problems with SBO in general, SBArchOpt additionally implements connections to other SBO libraries, including BoTorch [229], Trieste [245], HEBO [217], SEGOMOE [246] and SMARTy [247].

Finally, to support SAO algorithm development, SBArchOpt contains a library of *test problems* consisting of the test problems developed in this work (see Section 2.3.1) and many other problems from literature and pymoo's test problem database. The library features various combinations of continuous or mixed-discrete, (non-)hierarchical, single-objective or multi-objective, (un)constrained problems, with or without hidden constraints.

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

2.6. APPLICATION CASE I: JET ENGINE ARCHITECTURE OPTIMIZATION

This application case presents the design of a jet engine architecture solved by the developed BO algorithm. As discussed in Section 2.3.1, the jet engine problem has been implemented as a benchmark problem for testing SAO algorithms.

The benchmark problem is defined using a jet engine optimization testing framework implemented as part of this work, the purpose of which is to provide a flexible way to define more or less difficult optimization problems [11]. Figure 2.7 provides an overview of the framework. The user defines the optimization problem by selecting from available architectural choices (that define the design variables) and metrics, and by inputting the flight conditions and power offtakes to size the engine for. Available architectural choices include whether to define a turbofan or turbojet architecture, the number of compressor and turbine stages, the use of intercooling and inter-turbine burning, and where to apply bleed and power offtakes.

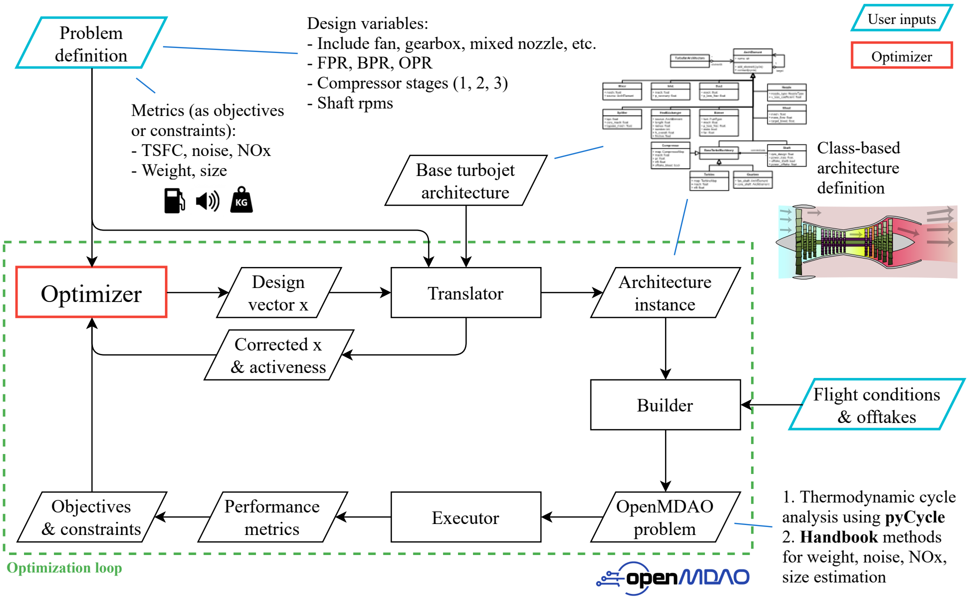

Problem definition
Design variables:
- Include fan, gearbox, mixed nozzle, etc.
- FPR, BPR, OPR
- Compressor stages (1, 2, 3)
- Shaft rpms
User inputs
Optimizer
Metrics (as objectives or constraints):
- TSFC, noise, NOx
- Weight, size
Base turbojet architecture
Class-based architecture definition
Optimizer
Design vector x
Translator
Architecture instance
Corrected x & activeness
Builder
Flight conditions & offtakes
Objectives & constraints
Performance metrics
Executor
OpenMDAO problem
1. Thermodynamic cycle analysis using pyCycle
2. Handbook methods for weight, noise, NOx, size estimation
Optimization loop
openMDAO

Figure 2.7: 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.

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 [248] problem that performs thermodynamic cycle analysis and engine sizing using pyCycle [249], 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[5].

Problem Instance Selection In this demonstration, the following problem formulation is used (available in SBArchOpt as SimpleTurbofanArch):

$$\begin{array}{rll}
\text{minimize} & \text{TSFC} & \\
\text{w.r.t.} & \text{IncludeFan} \in \{\text{False}, \text{True}\} & \\
& \text{if IncludeFan} = \text{True}: & \\
& \quad 2.0 \le \text{BPR} \le 12.5 & \\
& \quad 1.1 \le \text{FPR} \le 1.8 & \\
& \quad \text{MixedNozzle} \in \{\text{False}, \text{True}\} & \\
& \quad \text{IncludeGearbox} \in \{\text{False}, \text{True}\} & \\
& \quad \text{if IncludeGearbox} = \text{True}: & \\
& \quad\quad 1.0 \le \text{GearRatio} \le 5.0 & \\
& 1.1 \le \text{OPR} \le 60.0 & \\
& n_{\text{shafts}} \in \{1, 2, 3\} & \\
& \text{if } n_{\text{shafts}} > 1: & \\
& \quad 0.1 \le \text{PR}_{\text{factor,i}} \le 0.9 & i = 2, \ldots, n_{\text{shafts}} \\
& 1000 \le \text{RPM}_i \le 20000 & i = 1, \ldots, n_{\text{shafts}} \\
& \text{PowerOfftake} \in \{1, \ldots, n_{\text{shafts}}\} & \\
& \text{BleedOfftake} \in \{1, \ldots, n_{\text{shafts}}\} & \\
\text{subject to} & M_{\text{jet}} \le 1.0 & \\
& \text{PR}_{\text{factor,sum}} \le 0.9 & \\
& \text{PR}_{\text{max,i}} \le 15.0 & i = 1, 2, 3
\end{array}$$

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 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 $\text{PR}_{\text{factor,i}}$ 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 number of shafts n_{shafts}.

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

In total, the problem features 15 design variables, of which 9 are continuous and 6 are discrete (3 integer and 3 categorical). Of the design variables, 9 are conditionally active and 2 are value-constrained: the problem therefore requires both correction and imputation operations in order to ensure design vectors are valid. The problem features 5 design constraints, constraining the output jet Mach number M_{jet} and pressure ratio distributions over the selected compressor stages ($\text{PR}_{\text{factor,sum}}$ and $\text{PR}_{\text{max,i}}$). The underlying thermodynamic cycle analysis and sizing code does not always converge, leading to a hidden constraint with a failure rate (FR) of approximately 50% in a random DoE.

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 $\text{IR}_d = 216/70 = 3.1$ (Eq. (2.1)). The continuous imputation ratio $\text{IR}_c = 1.26$ (Eq. (2.2)), 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 $\text{IR} = 3.89$ (Eq. (2.3)). The correction ratio $\text{CR} = 2.10$ (Eq. (2.6)), which leads to correction ratio fraction $\text{CRF} = 55\%$ (Eq. (2.7)). Thus, a little over half of the design space hierarchy is due to value constraints (i.e. the need for correction). The max rate diversity MRD of the problem is 60%, stemming from the fact that 20% of the design vectors is made up by the turbojet architectures (IncludeFan = False) and the remaining 80% by the turbofan architectures (IncludeFan = True).

Optimization Results The BO algorithm is executed 24 times with $n_{\text{doe}} = 113$, $n_{\text{infill}} = 187$ (a budget of 300 evaluations), and $n_{\text{batch}} = 4$. The algorithm is executed for Repair integration (no hierarchy information exposed, however the repair operator is available), Hierarchical Sampling (therefore a non-hierarchical GP is used during the optimization), and Activeness (both hierarchical sampling and a hierarchical GP is used); refer to Table 2.11 for an overview. The hierarchical sampling configuration is executed for both the RFC and MD GP PoV predictors; all others only with the RFC predictor. NSGA-II was already able to deal with hidden constraints, however for completeness BO results are compared against NSGA-II results. NSGA-II is executed with repair operator available and using hierarchical sampling.

Figure 2.8 presents the results of the jet engine optimization for NSGA-II and the BO algorithm with the three compared hierarchy integration strategies. Table 2.19 presents achieved median optimal TSFC and ΔHV regret values. The BO algorithm configurations find results within 0.3% of each other with a budget of 300 function evaluations. NSGA-II is able to find the same result with 3250 evaluations: 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 2.19). 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.

Figure 2.9 presents a comparison between two different PoV predictors. It shows that the MD GP PoV predictor slightly outperforms the RFC PoV predictor, resulting in a slightly better TSFC at a better ΔHV regret.

Table 2.19: Comparison of median optimal TSFC (minimization) values achieved for the jet engine problem. Refer to Table 2.11 for comparison of hierarchy integration strategies. ΔHV regret was not available for NSGA-II with 3250 evaluations.

Algorithm	Hierarchy integration	PoV Predictor	N_{fe}	TSFC [g/kNs]	ΔHV regret
BO	Activeness	RFC	300	6.633	0.91
BO	Hierarchical sampling	RFC	300	6.653 (+0.30%)	0.86 (-4.8%)
BO	Hierarchical sampling	MD GP	300	6.635 (+0.03%)	0.67 (-27%)
BO	Repair	RFC	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%)	–

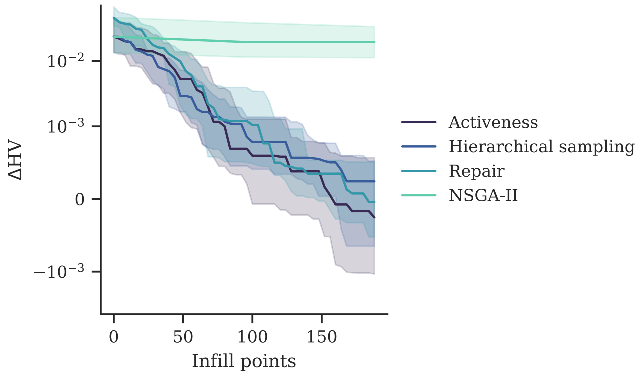


Figure 2.8: Comparison of NSGA-II to the BO algorithm with different levels of hierarchy integration (see also Table 2.11): Repair (no hierarchy information; repair operator available), Hierarchical sampling (no hierarchical GP) and Activeness (hierarchical sampling and hierarchical GP). ΔHV represents the distance to the known Pareto front (Eq. (E.1)). The bands around the lines represent the 50 percentile range around the median over 24 algorithm runs.

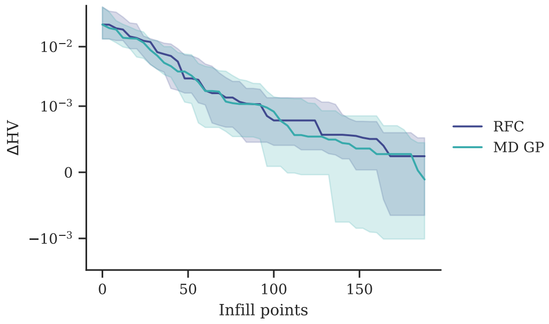


Figure 2.9: Comparison of two PoV predictors (see also Table 2.15) for dealing with hidden constraints in the BO algorithm (executed with "Hierarchical sampling" integration).

2.7. CHAPTER CONCLUSIONS

In this chapter, developments to optimization algorithms for solving SAO problems have been presented. As optimization algorithms, mainly Multi-Objective Evolutionary Algorithms (MOEAs; represented by NSGA-II in tests) and Bayesian Optimization (BO) algorithms are considered for solving SAO problems: BO if evaluation is expensive, MOEAs if not. Section 2.1 presents the non-hierarchical basis of the further-developed BO algorithm, featuring a mixed-discrete GP, an ensemble of infill criteria, and a constraint handling strategy. Section 2.2 defines several metrics for quantifying various aspects of design space hierarchy:

- imputation ratio IR, correction ratio CR, and correction fraction CRF for quantifying the impact of the need for imputation and correction for finding valid design vectors; and
- max rate diversity MRD for quantifying the behavior that some values of discrete design variables occur much more often than other values.

To support algorithm development, three collections of test problems are presented in Section 2.3.1: multi-stage launch vehicle problems, GNC problems (adopted from [39]), and jet engine architecture problems. Then, the main contributions of this research to improve performance of MOEAs and BO algorithms for SAO are presented:

- The development of a sampling algorithm for hierarchical design spaces to ensure no region in the design space is over- or under-represented in Section 2.3.2. The hierarchical sampling algorithm works by separating valid discrete design vectors into groups by active design variables x_{act}, uniformly sampling from these groups, and using Sobol' sampling to assign values to continuous variables.
- An investigation into whether problem-agnostic correction improves optimization performance compared to problem-specific greedy correction in Section 2.3.3, showing that it does not necessarily improve performance and thus problem-specific greedy correction is sufficient.
- An investigation into integration of hierarchy information into optimization algorithms in Section 2.3.4, showing that integrating more information (returning the corrected design vector and activeness information from the evaluation function, making correction and imputation available as a standalone repair operator, exposing which design variables are conditionally active, and making $x_{\text{valid,discr}}$ available) leads to better optimizer performance.
- The development of a strategy for dealing with hidden constraints in BO in Section 2.4. The strategy involves training an additional model that calculates the Probability of Viability (PoV), which is used in the infill search process to steer infill towards viable regions of the design space. Mixed-discrete GP and a Random Forest Classifier (RFC) models perform best.

All benchmark problems and developed optimization algorithms have been made available open-source in the SBArchOpt library, presented in Section 2.5. The developed algorithms are demonstrated in Section 2.6 by a jet engine architecture optimization problem, showing:

- the definition of a flexible jet engine architecting framework for defining SAO benchmark problems;
- the application of NSGA-II and the BO algorithm to a single-objective instance of that test problem; and
- optimization results showing that BO can be used to effectively solve SAO problems, including those subject to hidden constraints, with a significant reduction in function evaluations (92% for this application case) needed compared to evolutionary algorithms like NSGA-II.

To summarize, the BO algorithm developed in this chapter uses:

- hierarchical, mixed-discrete GP models;
- ensemble infill criteria with a sequential-optimization procedure for batch infill generation for single- and multi-objective optimization problems;
- a design constraint handling approach using constraint function mean prediction;
- a hierarchical sampling algorithm that groups valid discrete design vectors by active design variables x_{act}, to deal with rate diversity effects (also applied to NSGA-II);
- several strategies for dealing with imputation ratio effects (also applied to NSGA-II): returning the corrected design vector and activeness information from the evaluation function, making correction and imputation available as a standalone repair operator, exposing which design variables are conditionally active, and making $x_{\text{valid,discr}}$ available; and
- a strategy for handling hidden constraints that uses a mixed-discrete GP for predicting the Probability of Viability (PoV).

With these results, the first sub-objective has been achieved:

1. Develop global evolutionary and Bayesian Optimization (BO) algorithms capable of efficiently solving SAO problems, by:
   (a) integrating information about the hierarchical design space into the optimization algorithms;
   (b) developing a sampling algorithm that explicitly takes the hierarchical nature of the design space into account;
   (c) investigating the impact of the correction algorithm on optimization performance; and
   (d) developing a strategy for solving problems with hidden constraints when using BO algorithms.

3 ARCHITECTURE DESIGN SPACE MODELING

CHAPTER CONTENTS

THIS chapter presents a function-based method for defining System Architecture Optimization (SAO) problems. This forms the second part of the objectives presented in Section 1.6, allowing the optimization problem to be modeled and executed. Here follows a short overview of the SAO optimization concept, and an introduction to the contents of this chapter.

As also discussed in Section 1.5.3, design variable hierarchy has as a result that a design vector $\boldsymbol{x}$ suggested by an optimization algorithm is not always *valid*, that is: a design vector where value constraints have been corrected for, and where inactive

This chapter is based on [1, 2, 5, 8].

design variables have been imputed with canonical values. Because only valid design vectors represent architecture instances, an architecture generation step is introduced into the SAO loop. The purpose of this step is twofold:

- The design vector $\boldsymbol{x}$ is made *valid* by correction and imputation.
- The design vector $\boldsymbol{x}$ is *transformed* into a system architecture instance.

The valid design vector, along with information about which design variable is active (the activeness δ), is communicated back to the optimization algorithm in accordance with Eq. (2.16).

Figure 3.1 presents the concept of the SAO loop, divided into the architecture generator and the architecture evaluator. The *Architecture Generator* combines the *Optimization Algorithm* with the following additional functionality:

- The *Encode Design Variables* step, which encodes architecture design choices (consisting of architectural choices and architecture-specific design parameters) in the SAO problem model as a set of design variables x.
- The *Generate Architecture* step, which transforms ("decodes") a design vector $\boldsymbol{x}$ (representing a specific point in the design space) into an architecture instance, and corrects and imputes the design vector in the process.
- The *Interpret Metrics* step, which interprets performance metrics as objectives or constraints in the context of the SAO problem.

The *Architecture Evaluator* implements the *Evaluate Architecture* step, which takes an architecture instance as input, and provides performance metrics as output. Note that this is different from "classical" optimization, where the input to the evaluation function is the design vector.

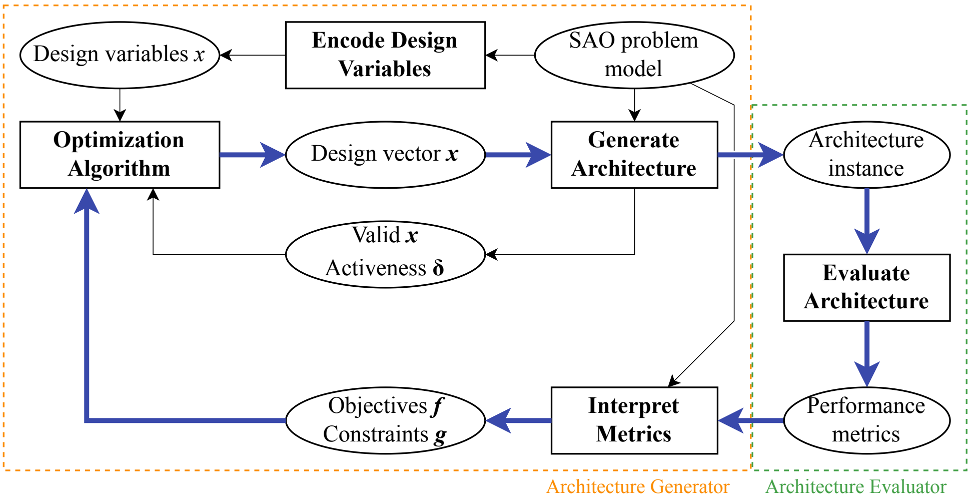


Figure 3.1: The SAO loop, highlighting the elements making up the Architecture Generator and Architecture Evaluator. The optimization loop is shown by bold blue arrows.

The SAO problem model provides all the information needed to perform the Architecture Generator steps. In this dissertation, these functionalities are developed and implemented in three layers:

1. Defining a way to model SAO problems for providing Architecture Generator functionality (Encode Design Variables, Generate Architecture, and Interpret Metrics) in the SAO loop.

   Section 3.1 presents the *Design Space Graph (DSG)*, which implements:

   - a directed graph for modeling hierarchical architecture choices and defining roles of performance metrics; and
   - algorithms for encoding architecture choices as design variables x and for decoding design vectors $\boldsymbol{x}$ into architecture instances.

2. Modeling SAO problems using systems engineering principles.

   Section 3.2 presents the *Architecture Design Space Graph (ADSG)*, which extends the DSG by defining node types and connection rules so that SAO problems can be defined using systems engineering principles.

3. Providing a user-friendly Graphical User Interface (GUI) for modeling.

   Section 3.3 presents *ADORE*, which provides:

   - a web-based GUI for modeling SAO problems using the ADSG; and
   - application Programming Interfaces (APIs) for connecting to architecture evaluation code and optimization algorithms.

   Section 3.4 presents the function-based process to use ADORE and the ADSG to model architecture design spaces.

Section 3.5 demonstrates the methodology by the architecture optimization of a hybrid-electric propulsion system. Section 3.6 compares optimization performance of SAO problems formulated with ADORE to manually-formulated SAO problems. The chapter is concluded in Section 3.7.

3.1. The Design Space Graph (DSG)

The Design Space Graph (DSG) implements mechanisms for modeling hierarchical architecture choices, defining design variables x from them, and for generating an architecture instance from a design vector $\boldsymbol{x}$. The process of defining design variables from architecture design choices (architectural choices and architecture-specific design parameters) is called "encoding". The process of generating an architecture instance from a design vector is called "decoding". The hierarchical structure between choices defines how choices select and connect nodes in a graph. Nodes represent elements (e.g. components, functions) of an architecture instance. The Python implementation of the DSG is available open-source as ADSG Core[1] (named as such because it represents the core mechanism of the ADSG).

Crawley et al. argue that the basic tasks of a system architect involve decomposing form and function, mapping function to form, specializing and characterizing form and function, and connecting form and function [39]. Selva et al. observe various reoccurring patterns in architecture choices [128], and map the architecting tasks to these patterns. Considering this, the Design Space Graph (DSG) has been developed based on the following assumptions:

1. The decomposing, mapping, specializing, and characterizing tasks involve the selection of architecture elements for inclusion in an architecture instance, and can therefore be referred to as *selection* tasks.
2. Architecture elements might select other architecture elements to be included, which might in turn activate other selection choices: there is a hierarchy between selection choices.
3. The *connecting* task is performed after selection tasks have been performed, and the elements to connect therefore depend on which elements have been selected in the selection tasks.

The selection and connection tasks are modeled using a directed graph with node and edge types defined in the **selection** and **connection** domains, respectively. The selection and connection domains are presented in more details in Section 3.1.1 and 3.1.2, respectively. Section 3.1.3 presents how these domains are combined to define the complete SAO problem.

[1] https://adsg-core.readthedocs.io/

3.1.1. The Selection Domain

The selection domain defines node and edge types for modeling choices involving the selection of architecture elements, and for modeling hierarchy between such selection choices. First the various node and edge types and their behavior are introduced, then the procedure for encoding and decoding selection choices is presented.

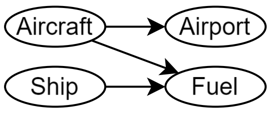
Selection Choice Modeling

Selection choices are modeled using three main concepts: derivation edges, start nodes, and selection choice nodes. Incompatibility constraints and choice constraints offer additional mechanisms for restricting selection choices.

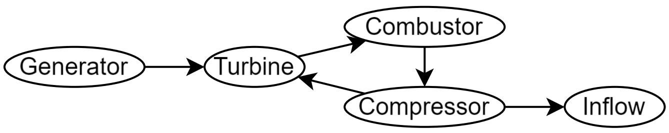
Derivation Edges A *derivation edge* is a directed edge that asserts that if the source node is included in an architecture instance, then the target node is also included in that same instance. A derivation edge can be interpreted as a "requires" relationship between two nodes, and can for example be used to represent a function to form mapping that is not subject to a choice (e.g. function induction [61]), or a static decomposition from a source (higher-level) function to one or more target (lower-level) functions.

Nodes can have any number of incoming and outgoing derivation edges. Multiple outgoing derivation edges mean that all target nodes are selected if the node is selected; multiple incoming derivation edges mean the node is selected if at least one of the source nodes is selected. Figure 3.2a shows an example of derivation edges. Aircraft derives both Airport and Fuel, so if Aircraft is selected both Airport and Fuel are selected as well. Fuel is derived by Aircraft and Ship, meaning that Fuel is selected if at least one of these is selected.

Cycles are allowed in the DSG: Figure 3.2b shows an example of this. If any of Turbine, Combustor, or Compressor are selected (in the example, the Turbine is selected by the Generator), all of them and their derived nodes (Inflow) are selected. The DSG therefore implements "level 4" (directed graph) of the hierarchical design space model classes as defined in Section 2.3.4.



(a) Aircraft derives Airport and Fuel; Ship derives Fuel.

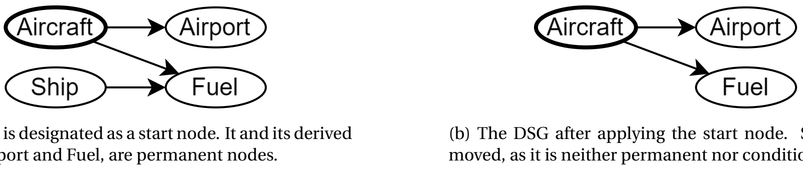


(b) Generator derives Turbine; Turbine, Combustor and Compressor derive each other in a cycle; Compressor derives Inflow.

Figure 3.2: DSG examples demonstrating derivation edges.

Start Nodes One or more nodes are designated as *start* nodes: these nodes and their derived nodes (not passing through selection choice nodes, see below) are present in all architectures and therefore are designated *permanent* nodes. Non-permanent nodes that have at least one path originating from a starting node (passing through one or more selection choice nodes) are *conditional* nodes. Nodes that are neither permanent nor conditional, are removed from the DSG after "applying" the start nodes. Start nodes are needed because cycles in the DSG may make it unclear where the derivation process starts.

Figure 3.3 shows an example of how start nodes behave. Aircraft is designated as a start node, and it derives Airport and Fuel: these three nodes are therefore permanent nodes. Ship is not a start node and is also not derived by one, therefore it is neither permanent nor conditional, and is removed from the DSG after applying the start node.



(a) Aircraft is designated as a start node. It and its derived nodes, Airport and Fuel, are permanent nodes.

(b) The DSG after applying the start node. Ship is removed, as it is neither permanent nor conditional.

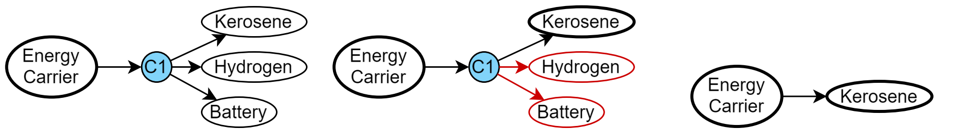
Figure 3.3: DSG example demonstrating a start node.

Selection Choice Nodes A *selection choice* node represents an architectural choice where one of the option nodes is selected. A selection choice node can only have one node connected by an incoming derivation edge: the *source node*. Choosing an option for a selection choice and modifying the DSG to reflect that choice is called *resolving* the selection choice, and consists of the following steps:

1. Mark the selected option node and its derived edges and nodes as *confirmed*. Include any encountered choice nodes, however exclude their option nodes (and derived nodes) from the set of confirmed nodes.
2. Mark the non-selected option nodes and their derived edges and nodes for removal, including choice nodes and their derived nodes. Exclude confirmed nodes and nodes that have multiple incoming derivation edges (if the incoming nodes are not also marked for removal). Include nodes and derived nodes targeted by incompatibility constraints that are triggered by confirmed nodes.
3. Remove the selection choice node and the nodes marked for removal from the DSG.
4. Connect the source node to the selected option node.

The DSG is considered to be in a new *state* after resolving each choice node. States that still contain choice nodes (except the initial state) are called *partial* states; states where there are no more choice nodes left are called *final* states. DSGs in a final state are also called architecture instances.

Figure 3.4a shows a selection choice between Kerosene, Hydrogen and Battery (the option nodes) as an Energy Carrier (the source node). Figure 3.4b shows Kerosene selected as the option (step 1) and Hydrogen and Battery marked for removal (step 2). Figure 3.4c shows the selection choice after it has been resolved: the nodes marked for removal are removed from the DSG (step 3) and Energy Carrier has been connected to Kerosene (step 4).



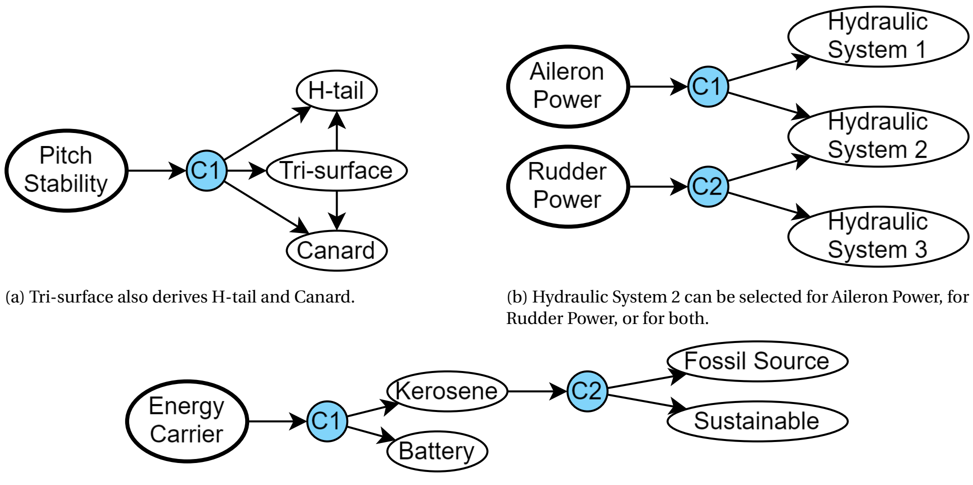
(a) Selection choice before resolving it. (b) The selected option node (Kerosene) is marked as confirmed; other option nodes (Hydrogen and Battery) are marked for removal. (c) The selection choice has been resolved.

Figure 3.4: DSG example demonstrating a selection choice node. Choice nodes are shown in blue.

Option nodes may derive each other, and since derived nodes of confirmed nodes are excluded from removal (step 2), option nodes may be selected as a derived node from another option node. Figure 3.5a shows an example of this: if the Tri-surface node is selected, H-tail and Canard will also be confirmed, however only an edge from Pitch Stability to Tri-surface will be established.

As nodes may be selected from multiple sources, nodes that are options of multiple selection choices can are confirmed if they are selected by at least one of the selection choices. Figure 3.5b shows an example: Hydraulic System 2 can be selected by either choices, and may co-exist with the other Hydraulic System nodes.

Choice nodes are *active* if they are or have been confirmed, *inactive* otherwise. This is relevant if not all choices are derived from start nodes, as is the case in the example in Figure 3.5c. There, the second choice is only active if Kerosene has been selected as Energy Carrier. Before that selection, or after selecting Battery, the second choice is inactive.



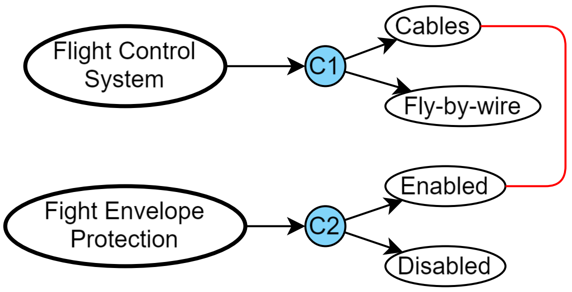
(a) Tri-surface also derives H-tail and Canard. (b) Hydraulic System 2 can be selected for Aileron Power, for Rudder Power, or for both.

(c) The second choice (C2) is only active if Kerosene is selected as Energy Carrier.

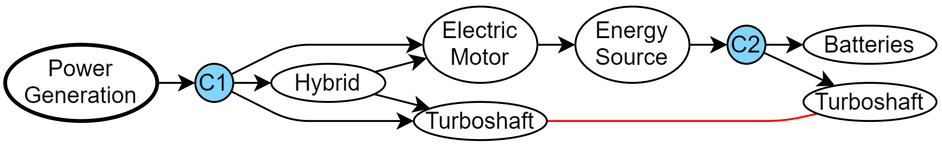
Figure 3.5: DSG examples demonstrating interactions within and between selection choices.

Incompatibility Constraints An incompatibility constraint is defined using an *incompatibility edge*: an undirected edge that asserts that if either of the two nodes is confirmed, the other node and its derived and deriving nodes are not. Note that also upstream nodes are affected: this is needed because if some node is removed by an incompatibility node, then it cannot be derived by any other node anymore, so to prevent violating the meaning of derivation edges the deriving nodes have to be removed as well.

Figure 3.6a shows an incompatibility constraint defined between two independent selection choices. It defines a Cable-based Flight Control System to be incompatible with an enabled Flight Envelope Protection system. If, however, Fly-by-wire is selected as basis for the Flight Control System, then the choice whether to enable Flight Envelope Protection is left free. Figure 3.6b shows an incompatibility constraint defined between different hierarchy levels. It defines that the two Turboshaft nodes may not exist together in an architecture instance. This means that if the Hybrid solution is chosen for Power Generation, only Batteries can be used as Energy Source for the Electric Motor. However, if only the Electric Motor is chosen for Power Generation, then the Energy Source choice is left free. Note that if the Turboshaft is chosen for Power Generation, the second choice (and therefore its option nodes) are removed from the DSG.



(a) Cables used as the basis for a Fight Control System are incompatible with Enabled (referring to the Flight Envelope Protection system).



(b) A Turboshaft used for Power Generation is incompatible with a Turboshaft used as Energy Source for an Electric Motor.

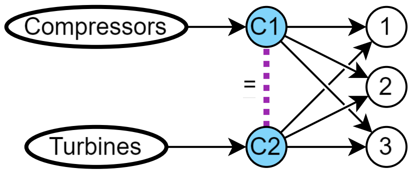
Figure 3.6: DSG examples demonstrating incompatibility constraints.

Choice Constraints *Choice constraints* can be applied to a set of selection choices to constrain the available options of choices in the set. This can be useful when defining choices in multiple locations in the design space that are actually linked, the number of compressors should for example be the same as the number of turbines in a turbofan engine. A choice can only be part of one choice constraint set. Four types of choice constraints are available:

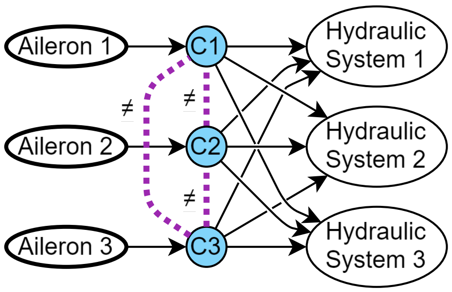
- *Linked* ("="): all choices are assigned the same option index, e.g. considering 2 choices with 3 options (A, B, and C): AA, BB, CC.
- *Permutations* ("≠"): all choices have a different option index, e.g. AB, AC, BA, BC, CA, CB.
- *Unordered combinations* ("≥"): choices have an equal or higher index than preceding choices, e.g. AA, AB, AC, BB, BC, CC.
- *Unordered non-replacing combinations* (">"): choices have a higher index than preceding choices, e.g. AB, AC, BC.

Note that "permutations" and "unordered non-replacing combinations" constraints require at least the same amount of options as the amount of choices: it is not possible to define permutations of 2 values for 3 choices for example.

Figure 3.7a shows an example of a linked choice constraint: there should be the same amount of Compressors and Turbines. Figure 3.7b shows an example of a permutations choice constraint: each Aileron should be assigned to a different Hydraulic System.



(a) The choices are linked: for both choices the same option is selected.



(b) A "permutation" constraint is applied: for each choice a different option should be selected.

Figure 3.7: DSG examples demonstrating choice constraints.

Figure 3.8 lists all node and edge types used for modeling selection choices.

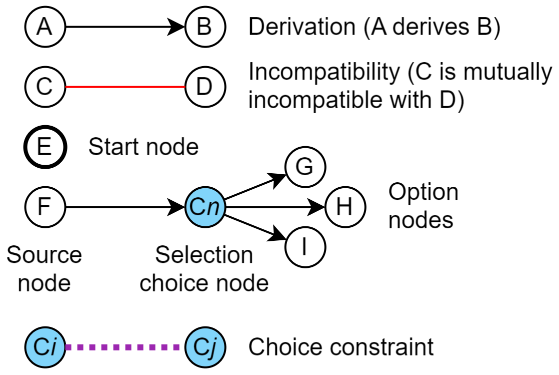


Figure 3.8: Legend showing all node and edge types involved in modeling selection choices.

Selection Choice Example Figure 3.9 shows an example DSG with 12 nodes (Nx), two selection choice nodes (Cx), N1 as the starting node, and two incompatibility edges. Table 3.1 lists for the same example which nodes are permanent or conditional, and for an enumeration of C1 and C2 options which nodes are confirmed or infeasible. There are 12 states for this DSG: 1 initial state (only containing permanent nodes), 3 partial states (C1 is resolved; C2 not) and 8 final states (including 2 infeasible and 6 feasible states). Starting nodes and nodes derived from any starting node not passing through a selection choice node are permanent: in this case N1, N2, N3 and C1. N0 is neither a permanent node (a starting nodes or derived by a starting node) nor a conditional node (derived by a selection choice), so it is never part of any architecture instance. Selection choice C1 is permanent and therefore always active, and its option node are N4, N5, N6, N12 and N13. N12 is incompatible with N2, and since N2 is permanent, N12 can never be selected. Selecting N4, N5 or N13 also selects N7 and therefore activates C2. Selecting N5 additionally selects N6, which selects N9. N9 is in a cycle with N10 and N8, meaning that if any of these nodes is selected, all of them are. If C2 is active, either N11 or N8 can be selected. Selecting the latter implies also selecting N9 and N10 due to the cycle. If N5 was selected for C1 and N11 for C2, both N8 and N11 are part of the final architecture.

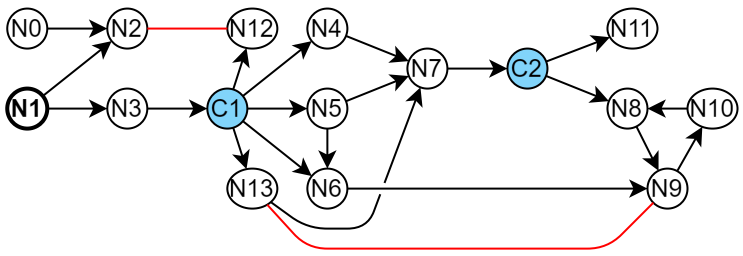


Figure 3.9: DSG example with generic nodes N and selection choice nodes C (shown in blue). Edges include derivation edges (black arrows) and incompatibility edges (red). Node N1 is the start node.

Table 3.1: Node status table for all possible combinations of choice options for the example shown in Figure 3.9. In the bottom part of the table, ✓ represents confirmed nodes in the associated (partial) architecture. × represents violated incompatibility constraints. Node names in the choice columns (C1 and C2) indicate the selected option node.

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

ENCODING AND DECODING SELECTION CHOICES

This section presents how selection choices are encoded as a set of discrete design variables x, and how a given design vector $\boldsymbol{x}$ is decoded into a selection of nodes.

Encoding is done by mapping each selection choice to a separate discrete design variable, with option nodes mapped to integer values. Table 3.2 shows the result of encoding the selection choices in Figure 3.9: it shows valid design vectors $x_{\text{valid,discr}}$, activeness information $\delta_{\text{valid,discr}}$, and node existence for all feasible architecture (see also Table 3.1). Selection choices C1 and C2 are mapped to design variables x_0 and x_1, respectively. Even though selection choice C1 has 5 options, the associated design variable x_0 only has 4 options, because N12 can never be selected as a feasible option. Selection choice C2 is not present in architecture 5, resulting in an inactive x_1 ($\delta_1 = 0$). This is also a good example of the difference between *declared* design space and *valid* design space sizes: the declared design space is given by the Cartesian product of design variable options, in this example $n_{\text{declared}} = 4 \cdot 2 = 8$, whereas the valid design space is given by the number of feasible architectures, in this example $n_{\text{valid}} = 6$. This difference is quantified by the imputation ratio IR_d, see Section 2.2.1.

Table 3.2: For each architecture in Table 3.1, the associated design vector x and activeness information δ. Selection choices C1 and C2 are mapped to x_0 and x_1, respectively.

	Design vector		Activeness	
Architecture	x_0	x_1	δ_0	δ_1
1	0	0	1	1
2	0	1	1	1
3	1	0	1	1
4	1	1	1	1
5	2	0	1	0
6	3	1	1	1

Two selection choice encoding algorithms have been developed as part of this work: the fast encoder and the complete encoder. The complete encoder results in design variable definitions with lower imputation ratios IR compared to the fast encoder, and enables the exhaustive identification of all valid design vectors. The complete encoder, however, requires significantly more computational resources (time, memory) than the fast encoder. When encoding the design space, therefore, first the complete encoder is tried. If some time or memory limit is reached, the fast encoder is used instead. A discussion of general principles follows. For more technical details about the implementation of the selection choice encoders see Appendix B.

Fast Encoder The *fast encoder* maps selection choices to discrete design variables, with option nodes mapped to integer values between 0 and n_{opts}-1 for each selection choice. For choice groups constrained by a linked choice constraint (i.e. all choices are assigned the same option index), only one discrete design variable is defined.

Decoding and correcting design vectors is performed in a greedy manner: starting from the initial DSG, confirmed (active) selection choices are assigned options from the given design vector. If for a given selection choice the requested option node is not available, the requested option node is corrected to the closest available option, thereby ensuring that the design vector represents a valid architecture. The process continues until the graph contains no more selection choices. Activeness information δ is provided by keeping track of which selection choices had options assigned during this process.

Complete Encoder The *complete encoder* exhaustively identifies all possible discrete design vectors $x_{\text{valid,discr}}$, associated activeness $\delta_{\text{valid,discr}}$ information, and node existence information. This enables using the hierarchical sampling and correction algorithms presented in Chapter 2 and the calculation of the number of valid architectures.

The complete encoder can additionally improve the discrete imputation ratio IR_d (Eq. (2.1)) of the problem by detecting *forced selection choices*: selection choices which only ever have one available option (when active) due to other selection choices disabling the other options. The selection choice C_i is forced if there exists a combination of selection choices $\{C_a, C_b, \ldots\}$ which fully defines the selected option index for C_i over all valid design vectors. Forced choices can be the result of incompatibility or choice constraints. Selection choices are encoded as discrete design variables by mapping each non-forced selection choice to a discrete design variable with available option nodes mapped to integer values.

Decoding and correcting design vectors is done by first checking if the requested design vector is valid (i.e. is part of $x_{\text{valid,discr}}$): if this is not the case, the closest valid design vector as measured by the Manhattan distance (Eq. (2.12)) is selected instead. The selected design vector is then used to derive an architecture instance graph from the initial DSG, by recursively resolving selection choices based on option node indices defined by the design vector.

3.1.2. The Connection Domain

The connection domain defines node and edge types for modeling choices involving the connection from a set of source nodes to a set of target nodes. First the various node and edge types and their behavior are introduced, then the procedure for encoding and decoding connection choices is presented.

Connection Choice Modeling

Connection choices offer a generic way to model source to target connection problems: problems where there are different connections that can be established between two sets of elements (sources and targets), and those connections are subject to constraints on the number of connections accepted by each source and target, and constraints on the total number of connections between each source and target element. Connection choices are modeled using three main concepts: connector nodes, connection edges, and connection choice nodes. Connector grouping nodes and exclusion edges may further restrict the number of options available for a given connection choice.

Connector Nodes Source and target nodes are represented using *connector nodes*, and specify their *connector constraint* as follows:

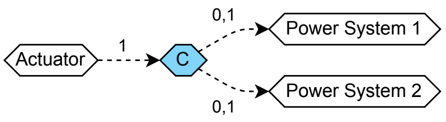
- The number of outgoing or incoming connection edges the connector node can accept. This can be specified as a list of numbers (e.g. 1, 2 or 3 connections: `1,2,3`), a lower and an upper bound (e.g. between 0 and 3, inclusive: `0..3`), or only a lower bound (e.g. 1 or more: `1..*`).
- Whether parallel connection edges originating from or targeting the connector node are allowed, shown using the ∥ symbol.

Connection Choice Nodes and Connection Edges A connection choice is defined by adding *connection edges* from one or more source connector nodes to a *connection choice node*, and from the connection choice node to one or more target connector nodes. Connector nodes can only be part of one connection choice, and connection choices in the DSG are independent of each other.

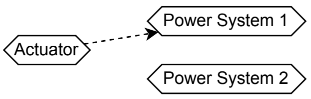
Connection choice options are made up of sets of valid connection edges: sets of edges from source to target connector nodes that adhere to all connector constraints. Choosing a set of valid connections for a connection choice and modifying the DSG to reflect that choice is called *resolving* the connection choice, and consists of the following steps:

1. Remove the connection choice node (and therefore also its incoming and outgoing connection edges).
2. Add valid connection edges from the set to the DSG.

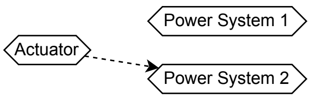
Figure 3.10a shows a connection choice with Actuator as the source connector node, and two Power Systems as target connector nodes. Actuator requires 1 outgoing connection; both Power Systems accept either 0 or 1 incoming connections. Effectively this means that Actuator can connect to either Power System. These two options are shown as resolved connection choices in Figures 3.10b and 3.10c.



(a) Actuator requires 1 outgoing connection; The Power Systems accept either 0 or 1 incoming connections.



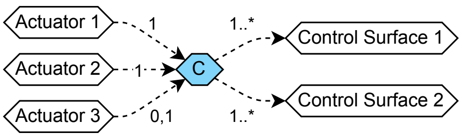
(b) Resolved connection choice (option 1): a connection to Power System 1.



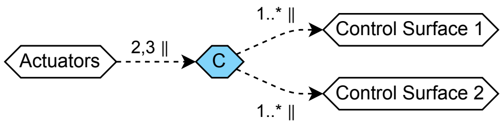
(c) Resolved connection choice (option 2): a connection to Power System 2.

Figure 3.10: DSG examples demonstrating a connection choice.

The property of whether parallel connections are allowed enables modeling connection choices where the order of connections is not relevant. To illustrate, Figure 3.11a shows a connection choice from 3 Actuators to 2 Control Surfaces. Each Actuator connects to either of the Control Surfaces independently of the others, however for the third Actuator the connection is optional (i.e. it can also be left "unused"). Figure 3.11b represents a similar connection choice, except that the order of connections does not matter: only the number of connections from the Actuators to the different Control Surfaces matters. Therefore, an architecture where Actuator 1 (2) connects to Control Surface 1 (2) is seen as a different architecture from one where Actuator 1 (2) connects to Control Surface 2 (1) in the first example, however they would be seen as the same architecture in the second example.



(a) A connection choice with no parallel edges: each Actuator connects to one of the Control Surfaces.



(b) A connection choice with parallel edges: the Actuators node connects to the Control Surfaces, allowing parallel connections between nodes. The ∥ symbol indicates that parallel connections are allowed.

Figure 3.11: DSG examples demonstrating connector constraints with or without parallel edges allowed.

Connector Existence Scenarios Connection choices are resolved after selection choices. Connector nodes behave just as any other node in the selection domain (Section 3.1.1), and may therefore exist conditionally. In the context of a given connection choice, each combination of connector node existence statuses defines a *connector existence scenario*. Since each existence scenario therefore includes a different set of source and/or target connector nodes, effectively each existence scenario defines a separate connection choice.

Figure 3.12 shows an example of this behavior. Selection choice C1 determines whether or not Actuator 3 exists. If Actuator 3 exists, then its outgoing connection has to be established; if it does not exist, then no outgoing connection is established. Effectively this means that for both existence scenarios (one where Actuator 3 exists; one where it does not) the connection choice is different, because different sets of valid connection edges are available as options. In terms of valid connection sets, Figure 3.12 shows the same behavior as Figure 3.11a. The difference is that in the former, the choice of whether or not a connection is made by the selection choice, whereas for the latter, the choice is made by the connection choice.

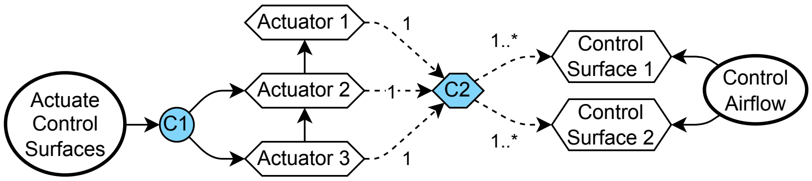


Figure 3.12: DSG example demonstrating conditional connector nodes. The existence of Actuator 3 depends on selection choice C1. If Actuator 3 does not exist, no outgoing connection is established.

Connector Grouping Nodes *Connector grouping nodes* can be used to represent cases where the order of connections is not relevant. This is done by grouping the connector constraints of multiple connector nodes into one overall connector constraint, and using the grouping node as the actual connector node in the connection choice. The use case is similar to parallel edges, except that the grouped connector constraint is modified for different node existence scenarios: if one or more of the grouped connector nodes does not exist, then their connector constraints are not included in the grouping.

Figure 3.13 shows an example of a connector grouping node. Outgoing connections from the Actuator connector nodes are grouped into one set of outgoing connections. If all Actuators exist, the grouping node provides 3 outgoing connections which are optionally parallel. If Actuator 3 does not exist, however, the grouping node connector constraint is updated to only 2 outgoing connections. Note that the DSG visualization shows the grouped connector constraint for the case where all connector nodes exist. In terms of possible established connections, Figure 3.13 shows the same behavior as Figure 3.11b. The difference is that in the former, the number of established connections is determined by the number of Actuator nodes and therefore by the selection choice, whereas for the latter, the number of connections is determined by the connection choice.

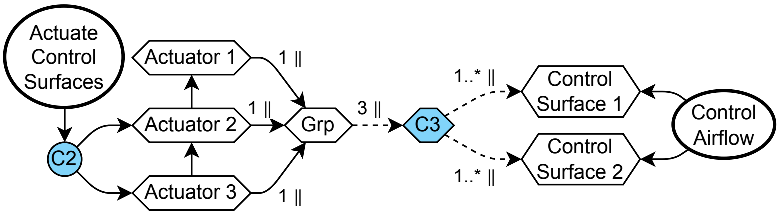


Figure 3.13: DSG example demonstrating a connector grouping node. The Actuator connector nodes are grouped into one set of outgoing connections. See Figure 3.8 for an overview of elements related to selection choices.

Exclusion Edges Finally, it is also possible to define combinations of source and target nodes that may not be connected, using *exclusion edges*. Figure 3.14 shows an example of a connection choice with an exclusion edge, preventing any connections from being established between the Backup Flight Computer and Sensor 1.

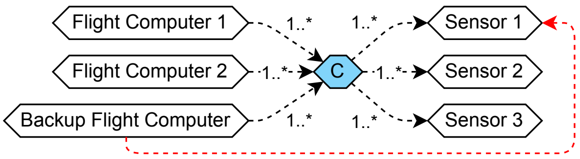


Figure 3.14: DSG example demonstrating an exclusion edge. The Backup Flight Computer may not be connected to Sensor 1.

Figure 3.15 lists all node and edge types used for modeling connection choices.

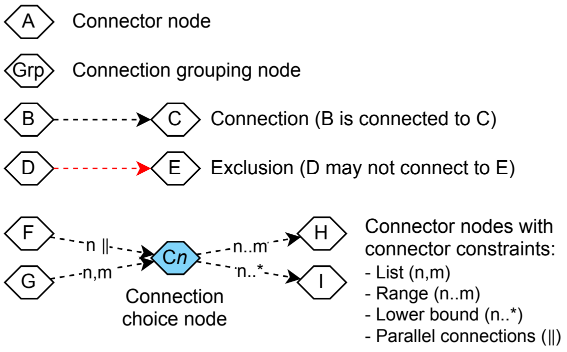

A
Connector node
Grp
Connection grouping node
B
C
Connection (B is connected to C)
D
E
Exclusion (D may not connect to E)
F
G
n ||
n,m
Cn
n..m
n..*
H
I
Connection choice node
Connector nodes with connector constraints:
- List (n,m)
- Range (n..m)
- Lower bound (n..*)
- Parallel connections (||)

Figure 3.15: Legend showing all node and edge types involved in modeling connection choices.

Connection Choice Example Figure 3.16 shows an example DSG with a connection choice (C2). C2 has as source nodes the connector grouping node Grp (which groups the connections of S1 and S2) and connector node S3, and target nodes T1 and T2. The connection edges, shown by dashed black arrows, display the connection constraints. The connection constraint of the grouping node Grp is aggregated from its underlying source nodes S1 and S2: each of these can have either 1 or 2 outgoing connections, which is aggregated to 2, 3 or 4 connections. The selection choice C1 determines whether S2 is confirmed (S1 is always confirmed). If S2 is not confirmed, only S1 remains and the aggregated connection constraint of Grp is modified to 1 or 2 connections (i.e. derived from S1 only). Table 3.3 enumerates all *valid connection sets* that exist for the example shown in Figure 3.16. It shows that if S2 exists, there are 3 possible sets of valid connection edges. If S2 does not exist, 5 valid connection sets exist.

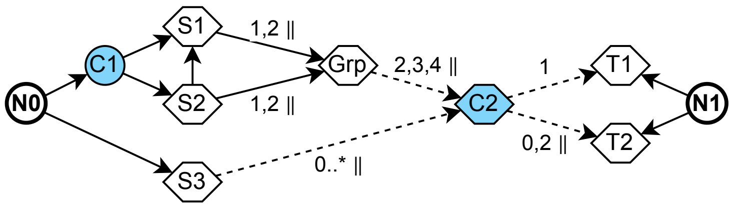


Figure 3.16: DSG example with generic nodes *N*, choice nodes *C* (blue). Connection (choice) nodes are shown as hexagons; connection edges as dashed lines. Nodes *N*0 and *N*1 are the start nodes. See Figure 3.8 for an overview of elements related to selection choices.

Table 3.3: Valid connection sets for the connection choice C2 shown in Figure 3.16. Each row is one set of valid connections. The left-side columns show the number of connections between pairs of source and target nodes. The right-side columns show the total number of connections incoming to or outgoing from each node.

	Connections between nodes				Total connections to / from node			
	Grp T1	Grp T2	S3 T1	S3 T2	Grp	S3	T1	T2
if S2 exists	0	2	1	0	2	1	1	2
	1	1	0	1	2	1	1	2
	1	2	0	0	3	0	1	2
if S2 does not exist	0	1	1	1	1	2	1	2
	1	0	0	0	1	0	1	0
	1	0	0	2	1	2	1	2
	0	2	1	0	2	1	1	2
	1	1	0	1	2	1	1	2

All architecture decision patterns identified by SELVA ET AL. [128] can be represented by connection choices, as shown in Table 3.4. Two additional combining patterns are defined: unordered (non-replacing) combining, which represents combining patterns where the order of option selection is not relevant. Additionally, parallel assigning patterns are added, because parallel (or multiple) connections were not considered in [128]. Due to its flexible formulation, also other connection choices can be modeled, such as shown in the example of Figure 3.16. Another example is presented in [23], where connection choices derived from safety regulations were modeled, specifying that each electric brake actuator should be connected to at least 2 independent electrical power sources. Another example are flight control system technology constraints used by BAUER ET AL. [250]: a constraint that each actuator should be connected to at least 1 but at most 2 flight computers, and a constraint that each control surface should have at least 2 ailerons assigned.

Table 3.4: Architecture decision patterns modeled using connection choices. n and m are independent integers equal to or greater than 1; n @ cc specifies the number of connector nodes (n) with connector constraint cc; (i, j) represents a connection from source i to target j; "$\|$" indicates that parallel connections are allowed.

Pattern	Variation	Source nodes	Target nodes	Excluded edges
Combining		n @ 1	m @ 0, 1	
Combining	Unordered	1 @ n $\|$	m @ 0..∗ $\|$	
Combining	Unordered non-replacing	1 @ n	m @ 0, 1	
Assigning		n @ 0..∗	m @ 0..∗	
Assigning	Injective	n @ 0..∗	m @ 0, 1	
Assigning	Surjective	n @ 0..∗ or 1..∗	m @ 1..∗	
Assigning	Bijective	n @ 0..∗ or 1..∗	m @ 1	
Assigning	Parallel	n @ 0..∗ $\|$	m @ 0..∗ $\|$	
Assigning	Parallel surjective	n @ 0..∗ or 1..∗ $\|$	m @ 1..∗ $\|$	
Partitioning		n @ 0..∗	m @ 1	
Partitioning	Covering	n @ 0..∗	m @ 1..∗	
Downselecting		1 @ 0..∗	m @ 0, 1	
Connecting		n @ 0..∗	n @ 0..∗	(i, j) if $i \geq j$
Connecting	Directed	n @ 0..∗	n @ 0..∗	(i, j) if $i = j$
Permuting		n @ 1	n @ 1	

ENCODING AND DECODING CONNECTION CHOICES

This section presents how connection choices are *encoded* as a set of discrete design variables x, and how a given design vector $\boldsymbol{x}$ is *decoded* into a set of valid connection edges. Here a discussion of the general principles follows; Appendix C provides more technical details about the implementation of the connection choice encoders.

Encoding connection choices is done by first constructing a *Connection Choice Formulation* (CCF) for each connection choice. A CCF consists of connector (grouping) nodes, connection edges, exclusion edges, and connector node existence scenarios for a given connection choice node.

Encoder Properties Different CCF patterns are best encoded as design variables using dedicated encoding grammars [80, 122, 128]. For example, the assignment pattern (i.e. assigning elements of one set to elements of another) is best encoded using a set of binary variables, each of which represents one combination of source and target nodes. The combination pattern (i.e. combining several choices with several options for each choice), however, is best encoded by each choice having one discrete variable with the number of available options. This is due to two effects discussed by SELVA [80]: bijectivity and non-degradedness.

Bijectivity relates to the difference in declared and valid design spaces: if this difference is large, it is more difficult to explore the design space for an optimization algorithm, because there is a low chance of generating a valid design vector when (randomly) searching the design space. In this work, this property is quantified using the discrete imputation ratio IR_d (Eq. (2.1)), the ratio between the number of declared discrete design vectors (i.e. the Cartesian product of all discrete variables) and the number of valid discrete design vectors. An imputation ratio of 1 indicates a one-to-one mapping between design vectors and architectures (i.e. bijectivity), whereas values higher than 1 indicate this is not the case. The higher the value, the larger the discrepancy. Refer to Section 2.2.1 for a more thorough definition.

Non-degradedness refers to the property of having a small (large) change in a design vector correspond to a small (large) change in what is represented by that design vector. Optimization algorithms depend on this property for local search and for building surrogate models for global optimization [251]. Local search namely assumes that if a small step is taken in the design space, the associated objective and constraint values also change by a small amount [49]. Surrogate models are based on the assumption that the closer a design vector lies to a design vector in the training database, the closer its associated objective and constraint values should be to the value in the training database [167].

In the case of connection choices, a design vector $\boldsymbol{x}$ represents a set of valid connections, which can be represented as an n_{src} x n_{tgt} connection matrix M, where n_{src} and n_{tgt} represent the number of source and target connector nodes involved in a given connection choice, respectively, and $M(i,j)$ represents the number of connections established between source node i and target node j. To quantify non-degradedness, the distance correlation Dcorr metric has been developed as part of this work. It correlates design vector distance to connection matrix distance, and is defined as:

$$\text{Dcorr} = \text{pearsonr}\left(\{d_{\text{manh}}(\boldsymbol{x},\boldsymbol{x}'),\ldots\},\{d_{\text{manh}}(M,M'),\ldots\}\right), \tag{3.1}$$

where pearsonr is the Pearson correlation coefficient, d_{manh} is the Manhattan distance, and $\boldsymbol{x}$ and $\boldsymbol{x}'$ are two randomly sampled valid design vectors with M and M' their corresponding connection matrices. The Pearson correlation coefficient measures the amount of linear correlation between two datasets, varying between -1 (perfect negative correlation) and 1 (perfect positive correlation) [252]. The Manhattan distance d_{manh} (Eq. (2.12)) is the sum of the differences of each term in the vector or matrix, thereby providing a good approximation of distance (or similarity) between discrete design vectors $\boldsymbol{x}$ or connection matrices M. Dcorr thus measures how much the Manhattan distance between design vectors (i.e. a change from one design vector to another) is correlated with the Manhattan distance between the associated connection matrices (i.e. the associated change in connection matrix). Dcorr = 1 indicates perfect correlation, meaning that a small (large) change in $\boldsymbol{x}$ indeed leads to a similarly small (large) change in its associated M. Lower values indicate less correlation, and the value is cutoff below 0 such that the lowest value Dcorr can take is 0, indicating no correlation.

The goal is therefore for a given CCF, to select an encoder that minimizes IR_d and maximizes Dcorr.

Encoder Classes The following classes of encoders have been developed as part of this work:

- *Pattern-specific* encoders: for each of the patterns in Table 3.4, an encoder can be defined that encodes the CCF according to optimal IR$_d$ and Dcorr behavior.

  Encoding is lossless, so for a given design vector $\boldsymbol{x}$ the associated connection matrix M can be reconstructed directly, which is how decoding is implemented.

- *Eager* encoders: encoders that define a more generic way of encoding and decoding connection choices compared to pattern-specific encoders. Encoding is done by first enumerating all valid M, denoted as $M_{\text{all,valid}}$, and then mapping these matrices to unique design vectors (M to $\boldsymbol{x}$ mapping), from which then design variables x are deduced.

  Decoding $\boldsymbol{x}$ into M is done by reverse lookup (i.e. looking for an existing $\boldsymbol{x}$ and obtaining the associated M) on the previously defined mapping, because the mapping may be lossy and therefore it is not possible to directly decode $\boldsymbol{x}$ into M. If no matching $\boldsymbol{x}$ is found, it is corrected by greedy correction: starting from the left, correct the design variables one-by-one to ensure their value is valid (i.e. matches at least one $\boldsymbol{x}$ in the mapping).

- *Lazy* encoders: these work without $M_{\text{all,valid}}$, and can therefore be used if determining $M_{\text{all,valid}}$ is not possible due to time or memory limits. Encoding is done directly using the CCF.

  Encoding is lossless, so decoding is done by directly reconstructing M from $\boldsymbol{x}$. If the resulting M is not valid, $\boldsymbol{x}$ is corrected in a trial-and-error manner: $\boldsymbol{x}$ is repeatedly modified until a valid M is found.

- *Ordinal* encoders: encoders that simply map $M_{\text{all,valid}}$ to ordinal indices, and encode these indices as integers or a numeral system with some base (e.g. binary variables for base 2).

  Decoding is done by index lookup, clipping the index if it is out-of-bounds.

Table 3.5 provides an overview of the different classes of connection choice encoders and their properties. It lists the encoder classes in the order of decreasing preference:

1. Pattern-specific encoders have the highest preference, because they do not need $M_{\text{all,valid}}$, have good IR$_d$ and Dcorr by definition, and do not depend on lookup or trial-and-error procedures for decoding and correction. They are therefore the fastest encoders with the lowest memory usage.
2. Eager encoders are preferred if the CCF matches no pattern-specific encoder. Eager encoders have fast decoding and correction times due to the full availability of $M_{\text{all,valid}}$.
3. Lazy encoders can be used if eager encoders run into time or memory limits when determining $M_{\text{all,valid}}$. Lazy encoders have low memory usage, however are slower in design variable correction.

4. Ordinal encoders are used as a fallback if all else either fails or produces a combination of a high IR_d and low Dcorr. Ordinal encoders are the least preferred, however, because of their by-definition low Dcorr.

Table 3.5: Classes of connection choice encoders and their properties, in order of decreasing preference from left to right. Abbreviation: CCF = Connection Choice Formulation.

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

Encoding Procedure The overall connection choice encoding procedure starts by determining all node existence scenarios for the connector nodes involved in a given connection choice. If the complete selection choice encoder is used (see Section 3.1.1), this can be done accurately. However, if the fast selection choice encoder is used, node existence is not available and it has to be assumed that any combination of existence for all of the connector nodes is possible. This assumption might lead to a less efficient connection choice encoding, for example because it prevents the use of a pattern-specific encoder.

Once node existence scenarios are determined, the CCF is obtained and encoders can be selected based on the preference order as discussed before. The selection algorithm compares IR_d (Eq. (2.1)) and Dcorr (Eq. (3.1)) for all relevant encoders, and attempts to select the encoder with $\text{IR}_d = 1$ and Dcorr = 1. If that is not possible, the encoder with the highest possible Dcorr below some IR_d threshold is selected. The selection algorithm is presented in more details in Section C.5.

3.1.3. The System Architecture Optimization Problem

This section discusses additional node types, which together with the selection and connection domains are combined into the complete SAO problem definition.

Design Variable Nodes Next to selection and connection choices, it is also possible to define design variables directly using *design variables nodes*. This can for example be used to model architecture-specific design parameters. Continuous design variables are defined by a lower and upper bound (inclusive); discrete design variables by a list of option values. Design variable nodes are subject to selection choices just as any other node, and can therefore exist conditionally. If a design variable node is not part of a DSG, the design variable is inactive. Design variable nodes can be constrained using choice constraints (see Section 3.1.1). For continuous design variables only the "linked" constraint is available.

Figure 3.17 shows an example containing design variable nodes. The Turbofan derives the Bypass Ratio continuous design variable (value between 5 and 15, inclusive). The Turboprop derives the Number of Blades discrete design variable (value is either 3, 4, 6 or 8). The design variable nodes are derived by conditional nodes, therefore the design variable nodes are also conditional.

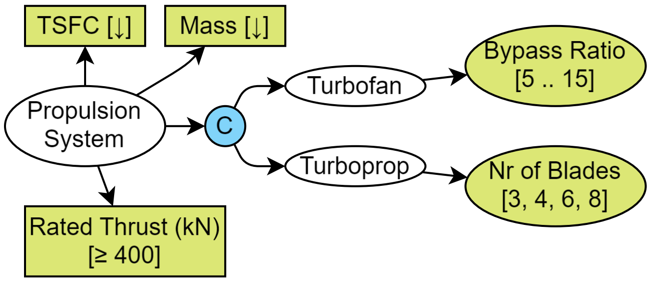


Figure 3.17: DSG example demonstrating design variable and metric nodes (both shown in gold). Turbofan and Turboprop each have a design variable associated: Bypass Ratio (continuous) and Number of Blades (discrete), respectively. The Propulsion System has three metrics associated: mass and TSFC (Thrust-Specific Fuel Consumption) as minimization objectives (indicated by the downward pointing arrows) and Rated Thrust as inequality constraint.

Metric Nodes The optimization problem definition is completed by defining performance metrics using *metric nodes*. A metric nodes represents an output of the architecture evaluation and can be used in three ways:

- As *objective* f, representing a minimization or maximization goal. A metric node can only be used as objective if it is permanent, because otherwise it is not possible to compare the performance of all architecture instances.
- As *inequality constraint* g, representing a value that should be above (greater than or equal) or below (lower than or equal) some threshold. Metrics used as inequality constraints can be conditional: if the node is not part of an architecture, the constraint is assigned the value of the threshold, meaning that in the context of the optimization problem the constraint is effectively considered to be satisfied.
- As a *generic metric*. The metric then plays no role in the context of the optimization problem, however they can be useful for verifying the correct behavior of the architecture evaluation code.

The example shown in Figure 3.17 also contains metric nodes, all derived by the Propulsion System: Mass and TSFC (Thrust-Specific Fuel Consumption) as minimization objectives, and Rated Thrust as an inequality constraint.

To summarize, an SAO problem can be modeled with the DSG using:

- derivation edges, start nodes, selection choice nodes, incompatibility constraints and choice constraints to define selection choices;

- connector (grouping) nodes with connector constraints, connection edges, exclusion edges and connection choice nodes to define connection choices;
- generic design variable nodes and choice constraints to define additional continuous or discrete design variables; and
- metric nodes to define output metrics, optionally used as objectives or inequality constraints.

Encoding and Decoding the DSG The set of design variables x that encode a given DSG is given by the combination of encoded selection choices (see Section 3.1.1), connection choices (see Section 3.1.2), and design variable nodes. The procedure for decoding a given design vector $\boldsymbol{x}$ into an architecture instance then is as follows:

1. Decode selection choices: correct and decode the partial design vector associated to the selection choices, track activeness information, and resolve selection choices.
2. Decode connection choices: for each connection choice, use the associated connection choice encoder to correct and decode the partial design vector into a connection matrix M, then resolve the connection choice using the connection matrix.
3. Assign values to design variable nodes directly from the associated design variables, tracking which values were assigned to determine which design variables are active.
4. Impute inactive design variables: set inactive discrete variables to 0, inactive continuous variables to mid-bounds.

3.2. THE ARCHITECTURE DESIGN SPACE GRAPH (ADSG)

Nodes in the DSG represent elements that can be included in architecture instances, however by themselves they have no meaning. This section presents the Architecture Design Space Graph (ADSG): a database of node types and allowable connections that assigns meaning to the DSG nodes for use in a system architecting context. Node types are defined based on the architecting tasks defined by CRAWLEY ET AL. [39]:

- function-to-component (form) allocation and function decomposition;
- function and component characterization; and
- component connection.

Function-to-Component Allocation and Function Decomposition The following node types are defined for modeling function-to-component allocations:

- *Function* nodes (symbol: FUN), specifying what the system should do. Functions that act at the system boundary and deliver the main value to the system stakeholders are known as boundary functions [61], and are used as start nodes.

  Examples of functions are "generate power", "store energy", or "move air".

- *Component* nodes (symbol: COMP), representing elements of form. Each component node has at least one component instance node (symbol: INST) associated to it, representing the instantiation of that component. Components fulfill functions, represented by a derivation edge from the function to the component node. Components can derive the existence of additional functions by defining derivation edges to function nodes. This is called function induction, because it induces the existence of the derived functions due to the selection of the component [61].

  For example, the "generate thrust" function may be fulfilled by a "turbofan" component, which in turn might induce the "provide fuel" function.

- *Multi-fulfillment* nodes (symbol: MULTI), representing multiple components fulfilling a function simultaneously. Derivation edges are established from the function node to the multi-fulfillment node, and then from the multi-fulfillment node to the multiple component nodes.

  For example, in a hybrid-electric propulsion architecture the "generate shaft power" function may be fulfilled by both an "electric motor" and a "turboshaft" component.

- *Non-fulfillment* nodes (symbol: NOF), representing the fact that a function is left unfilfilled. This can be useful in the context of a selection choice where the choice is between fulfilling a function using some component, or leaving the function unfulfilled.

  For example, in a turbofan system the function "improve performance" may either be fulfilled by a "gearbox" component or left unfulfilled (i.e. connected to a non-fulfillment node).

Figure 3.18 shows function fulfillment and function induction by components: the boundary function GENERATE THRUST is fulfilled by the PROPELLER component. The PROPELLER component induces two functions: DECOUPLE RPM and GENERATE MECHANICAL POWER. GENERATE MECHANICAL POWER is either fulfilled by the ELECTRIC MOTOR component, the TURBOSHAFT component, or the HYBRID multi-fulfillment (which then derives both of the former components). DECOUPLE RPM is either fulfilled by the GEARBOX component or left unfulfilled (represented by the DO NOTHING non-fulfillment node). Note that each component also has one component instance node associated to it.

The function decomposition task can be modeled using:

- *Decomposition* nodes (symbol: DE), mapping one higher-level function to one or more lower-level functions. A decomposition means that the originating function performs the same as the combination of the target functions: the originating function is said to emerge from the target functions.

  For example, the function "control aircraft" can be decomposed into the "control pitch", "control yaw", "control roll", and "control thrust" functions.

- *Concept* nodes (symbol: CON), mapping one solution-neutral function to one (relatively) solution-specific function [39].

For example, the function "transport passengers" can be mapped to "transport passengers by air" when developing an airborne transportation system.

Functions are fulfilled by defining a derivation edge from a function node to a Function Derivation Node (FDN): a component, multi-fulfillment, non-fulfillment, decomposition, or concept node. If there are multiple outgoing derivation edges to FDN nodes, a "function fulfillment" selection choice node is inserted. Figure 3.18 contains two such selection choice nodes (shown in blue): for choosing how the DECOUPLE RPM and GENERATE MECHANICAL POWER functions are fulfilled.

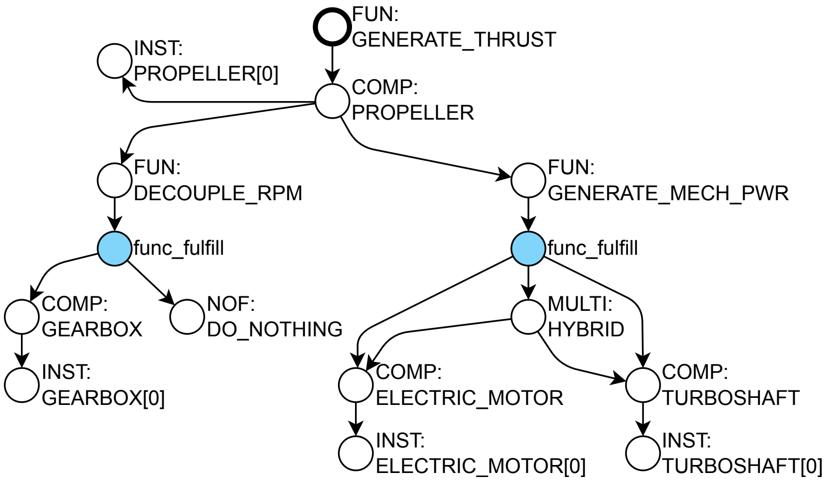


Figure 3.18: ADSG example demonstrating function fulfillment and induction by components, and usage of multi-fulfillment and non-fulfillment elements.

Function and Component Characterization Characterization includes architectural choices and design parameters (design variables) defined at the function-level or component-level:

- Design variable value allocation using design variable nodes (symbol: DV; see Section 3.1.3). Design variable nodes define either discrete or continuous design variables, and can be associated either to a function, a component, or a component instance node.

  Figure 3.19 shows a BYPASS RATIO design variable associated to the TURBOFAN component.

- Static input definition using *static input* nodes (symbol: INP). A static input node represents some value associated to a function, component or component instance that provides input to the evaluation, however is not free to change (i.e. will not be included as a design variable). This can be used to make assumptions or static component properties more explicit, compared to keeping them as input (or even hard-coded) variables inside the evaluation code.

Figure 3.19 shows a RATED THRUST static input associated to the GENERATE THRUST function.

- Metric definition using *metric* nodes (symbol: MET; see Section 3.1.3). Metric nodes can be associated to function, component or component instance nodes and represent outputs of the architecture evaluation. A metric node can additionally define a direction (maximization or minimization), which allows it to be used as an objective (symbol: OBJ), or a value threshold, allowing it to be used as an inequality constraint (symbol: CON).

  Figure 3.19 shows a TSFC (Thrust-Specific Fuel Consumption) minimization objective associated to the GENERATE THRUST function, and a MASS metric associated to the TURBOFAN component.

- Selecting the number of component instances. Each component instance is represented by a component instance node (symbol: INST). An instantiation choice is specified by adding a selection choice node that as options has grouping nodes which in turn derive the component instance nodes associated to the number of instances for that option.

  An example is shown in Figure 3.19: the TURBOFAN component is either instantiated 2 or 4 times, represented by the component instantiation selection choice node with two grouping nodes as options, of which the first selects the first 2 instance nodes, and the second selects all 4 instance nodes.

- Component attribute selection using *attribute* nodes (symbol: ATTR). An attribute node represents an assignment of one or more attribute values (symbol: VAL). The assignment choice is modeled using a connection choice (see Section 3.1.2).

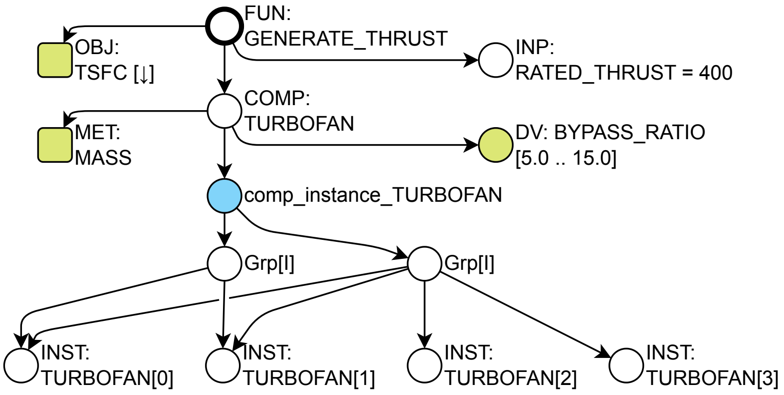


Figure 3.19: ADSG example demonstrating function and component characterization: a design variable node, a static input node, metric nodes, and a component instantiation choice. Choice nodes are shown in blue; design variable and metric nodes in gold.

Component Connection Component connection choices can be modeled using connection choices (see Section 3.1.2) from output port connectors (symbol: OUT) to input port connectors (symbol: IN). Port connector nodes are derived by component instances. The number of port connector nodes per component instance can also be a choice, modeled using a selection choice.

Figure 3.20 shows an example of a component connection choice. The CONTROL SURFACE and POWER DISTRIBUTION SYSTEM components both have two instances. The connection choice is defined using the ASSIGN port, with output connectors derived by the power distribution system instances and input connectors by the control surface instances. The port connectors define connection constraints: each power distribution connector can establish any number of connections; the control surface connectors can either receive 1 or 2 connections.

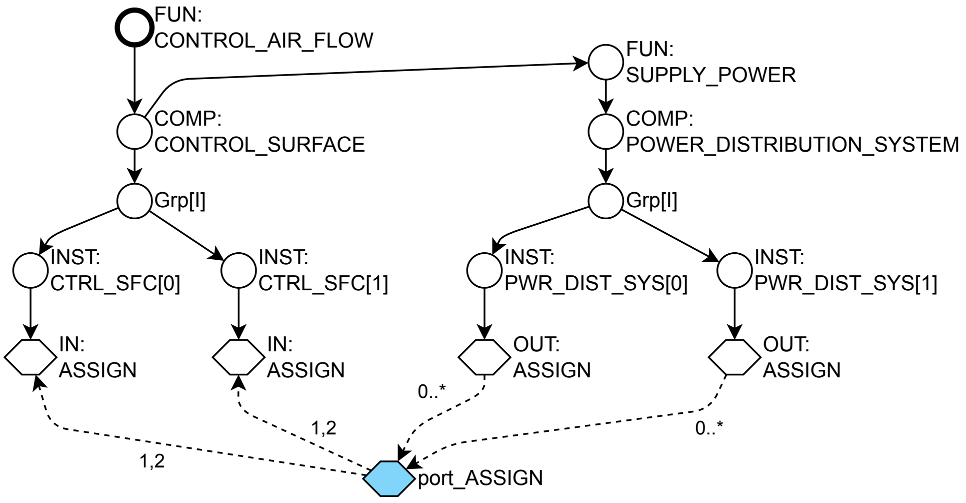


Figure 3.20: ADSG example demonstrating a component connection choice.

Table 3.6 lists all available ADSG node types and possible derivation edge connections to and from these node types.

Table 3.6: ADSG node types and allowed incoming/outgoing derivation edge connections. Abbreviations: FDN = function derivation node (node in the function allocation task other than the function node), F/M = function or multi-fulfillment node, QOI = Quantity of Interest (MET, DV, or INP), CCN = component characterization node (QOI or ATTR), C/I = COMP or INST.

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

3.3. ADORE Modeling and Optimization Environment

ADORE (Architecture Design and Optimization Reasoning Environment) is a Python tool developed as part of this work for defining and solving SAO problems. It provides:

- a web-based Graphical User Interface (GUI) for editing and inspecting an ADSG;
- Python and file-based interfaces for connecting to evaluation code; and
- interfaces for connecting to optimization algorithms.

An overview of these three aspects follows.

3.3.1. ADORE Models and Graphical User Interface

The architecture design space is defined in an ADORE model. During the modeling and optimization processes, in the background the ADSG is created from the ADORE model, and is used to deduce ADORE model behavior. Appendix D presents the interactions between the DSG, ADSG and ADORE in more details, and it shows how the SAO loop shown in Figure 3.1 is implemented.

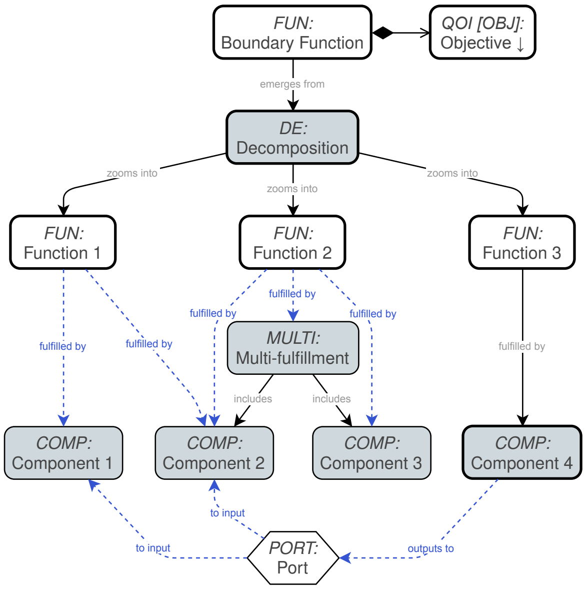
ADORE models represent the same concepts as the ADSG, however with several differences to improve clarity and manage complexity. Several model views are defined to reduce the number of items shown to the user at a given time:

- a system view showing functions, function derivation elements (components, decompositions, etc.), and ports;
- a component view showing component-level elements such as attributes; and
- a port view showing a port, its port connectors, and associated components.

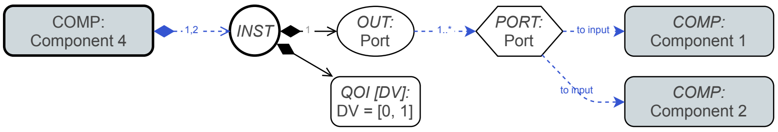
Metrics, design variables, and static inputs are defined using Quantities of Interest (QOIs), enabling flexible switching between input or output roles. This is useful, as during design space definition it may not be known yet whether some value is an input to or an output from the architecture evaluation process. The ADORE model also adds the capability for modeling subsystems: recursive groups of elements, where the number of subsystem instances may also be an architectural choice. Each subsystem instance then contains copies of the original elements, including choices, and copied choices are independent of each other. Figure 3.21 shows an example ADORE model.

The web-based GUI of ADORE is built-up of five pages to assist the architect in defining and executing SAO problems:

1. The design space editing canvas (shown in Figure 3.22), where the ADORE model is inspected and edited using the three views presented in Figure 3.21.
2. The external database list (shown in Figure 3.23), showing databases of external elements and how they are linked to ADORE model elements. External databases are optional, and their usage depends on the context that ADORE is applied in. For example, in the AGILE4.0 project external databases were used to link ADORE model elements to requirements [21].
3. The architecture design choices list (shown in Figure 3.24), which allows defining choice constraints (see Section 3.1), and shows design space statistics (e.g. encoder types, design space sizes, number of choices, and imputation ratio).



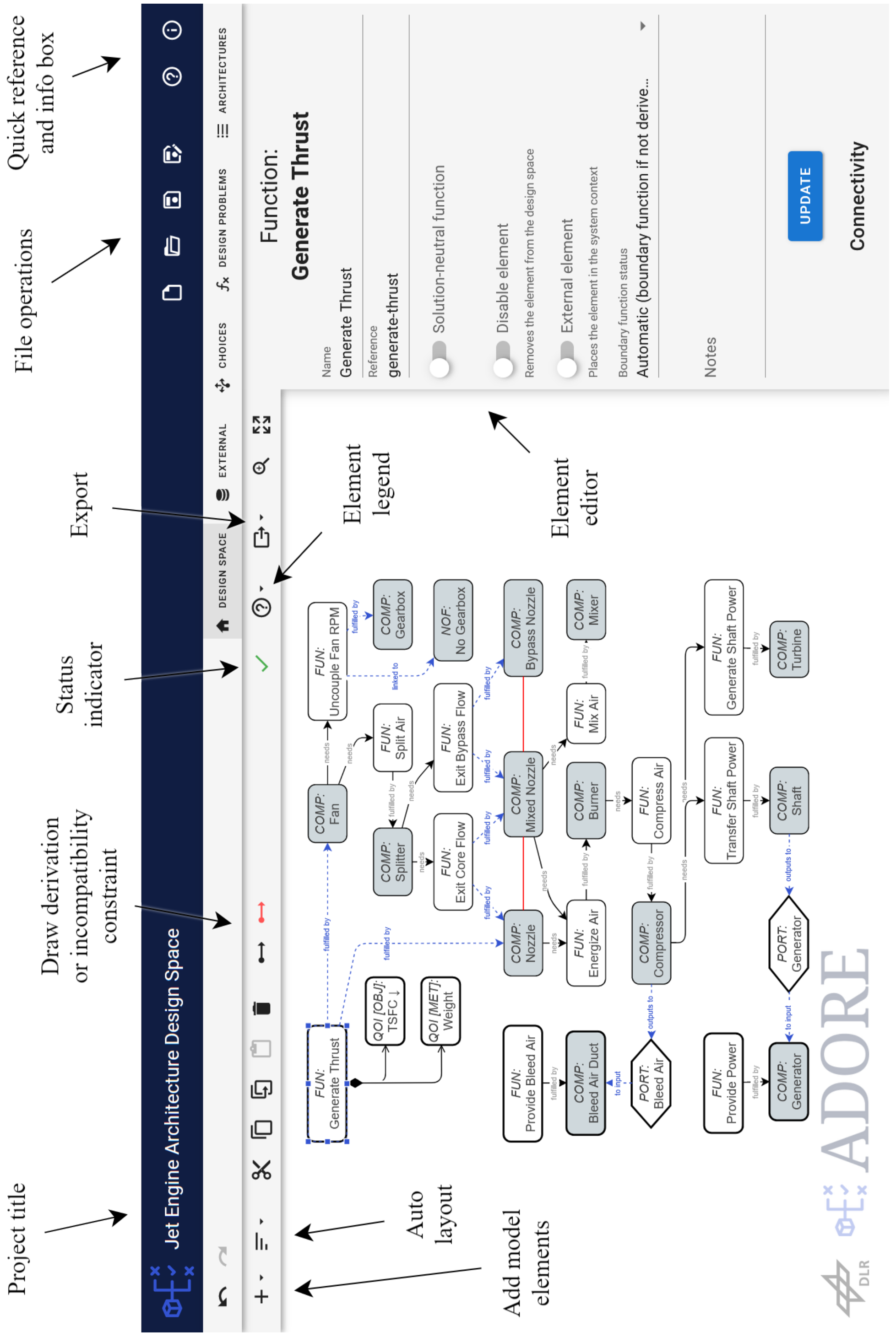
(a) System view



(b) Component view (Component 4)

Figure 3.21: ADORE model showing a design space with a boundary function (which is also the start function) with an objective, a decomposition into 3 lower-level functions, selection choices (2x function fulfillment, 1x component instantiation), instance-level design variables, and a port connection choice. Blue-dashed arrows indicate architectural choices.

4. The design problems list, where design (optimization) problems can be formally defined (shown in Figure 3.25). Their definition additionally allows fixing design variables to easily enable exploring only a sub-part of the design space. It also allows changing the output roles (objectives, constraints, or generic metrics).
5. The architecture instances list, showing all architecture instances that have been generated from the design space model, either as part of a design problem or by manual creation. Manually creating architecture instances is useful for verifying that the design space indeed represents the behavior that the architect intends, and for testing architecture evaluation code. Figure 3.26 shows the decisions editor for manually creating an architecture instance.



Function:
Generate Thrust

Name
Generate Thrust

Reference
generate-thrust

Solution-neutral function

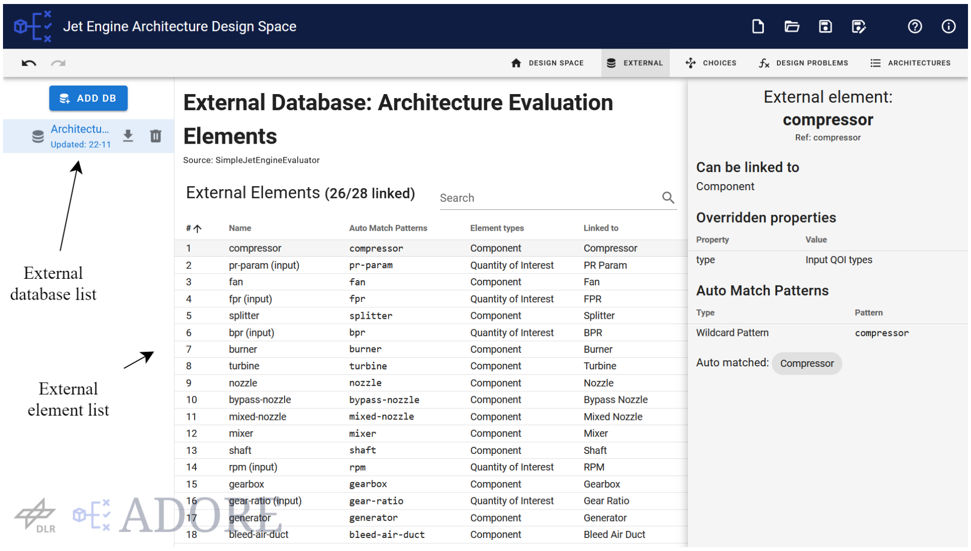
Disable element
Removes the element from the design space

External element
Places the element in the system context

Boundary function status
Automatic (boundary function if not derive...

Notes

Figure 3.22: Annotated overview of the web-based graphical user interface of ADORE, showing the design space canvas in system view.



External Database: Architecture Evaluation Elements

Source: SimpleJetEngineEvaluator

External Elements (26/28 linked)

#	Name	Auto Match Patterns	Element types	Linked to
1	compressor	`compressor`	Component	Compressor
2	pr-param (input)	`pr-param`	Quantity of Interest	PR Param
3	fan	`fan`	Component	Fan
4	fpr (input)	`fpr`	Quantity of Interest	FPR
5	splitter	`splitter`	Component	Splitter
6	bpr (input)	`bpr`	Quantity of Interest	BPR
7	burner	`burner`	Component	Burner
8	turbine	`turbine`	Component	Turbine
9	nozzle	`nozzle`	Component	Nozzle
10	bypass-nozzle	`bypass-nozzle`	Component	Bypass Nozzle
11	mixed-nozzle	`mixed-nozzle`	Component	Mixed Nozzle
12	mixer	`mixer`	Component	Mixer
13	shaft	`shaft`	Component	Shaft
14	rpm (input)	`rpm`	Quantity of Interest	RPM
15	gearbox	`gearbox`	Component	Gearbox
16	gear-ratio (input)	`gear-ratio`	Quantity of Interest	Gear Ratio
17	generator	`generator`	Component	Generator
18	bleed-air-duct	`bleed-air-duct`	Component	Bleed Air Duct

External element: **compressor**

Ref: compressor

Can be linked to

Component

Overridden properties

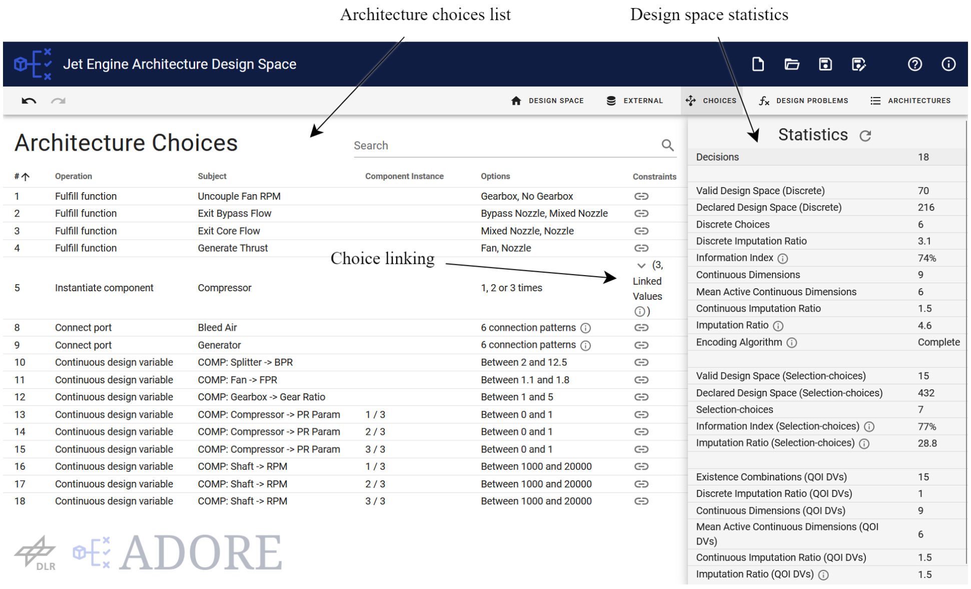
Property	Value
type	Input QOI types

Auto Match Patterns

Type	Pattern
Wildcard Pattern	`compressor`

Auto matched: Compressor

Figure 3.23: ADORE graphical user interface, showing external databases.



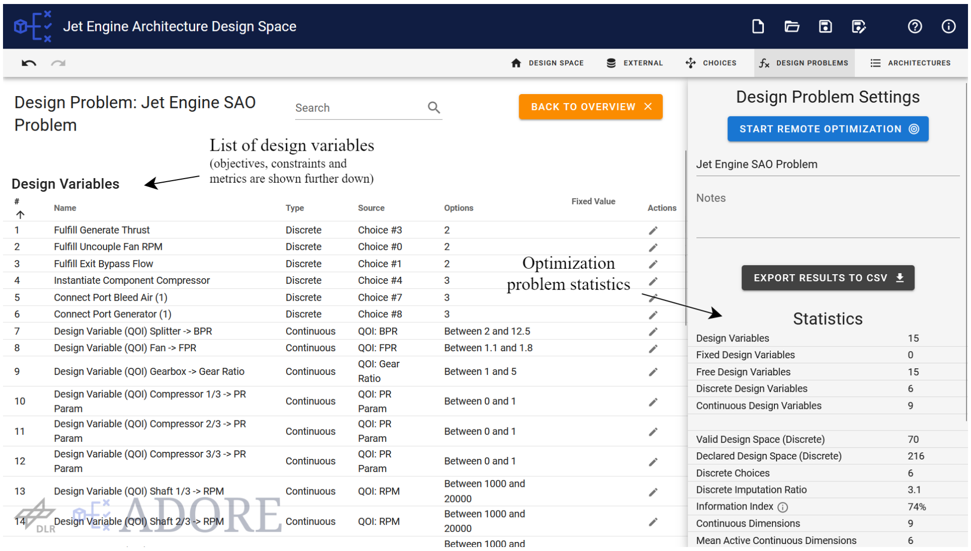
Architecture Choices

#	Operation	Subject	Component Instance	Options	Constraints
1	Fulfill function	Uncouple Fan RPM		Gearbox, No Gearbox	
2	Fulfill function	Exit Bypass Flow		Bypass Nozzle, Mixed Nozzle	
3	Fulfill function	Exit Core Flow		Mixed Nozzle, Nozzle	
4	Fulfill function	Generate Thrust		Fan, Nozzle	
5	Instantiate component	Compressor		1, 2 or 3 times	(3, Linked Values ⓘ)
8	Connect port	Bleed Air		6 connection patterns ⓘ	
9	Connect port	Generator		6 connection patterns ⓘ	
10	Continuous design variable	COMP: Splitter -> BPR		Between 2 and 12.5	
11	Continuous design variable	COMP: Fan -> FPR		Between 1.1 and 1.8	
12	Continuous design variable	COMP: Gearbox -> Gear Ratio		Between 1 and 5	
13	Continuous design variable	COMP: Compressor -> PR Param	1 / 3	Between 0 and 1	
14	Continuous design variable	COMP: Compressor -> PR Param	2 / 3	Between 0 and 1	
15	Continuous design variable	COMP: Compressor -> PR Param	3 / 3	Between 0 and 1	
16	Continuous design variable	COMP: Shaft -> RPM	1 / 3	Between 1000 and 20000	
17	Continuous design variable	COMP: Shaft -> RPM	2 / 3	Between 1000 and 20000	
18	Continuous design variable	COMP: Shaft -> RPM	3 / 3	Between 1000 and 20000	

Statistics

Decisions	18
Valid Design Space (Discrete)	70
Declared Design Space (Discrete)	216
Discrete Choices	6
Discrete Imputation Ratio	3.1
Information Index ⓘ	74%
Continuous Dimensions	9
Mean Active Continuous Dimensions	6
Continuous Imputation Ratio	1.5
Imputation Ratio ⓘ	4.6
Encoding Algorithm ⓘ	Complete
Valid Design Space (Selection-choices)	15
Declared Design Space (Selection-choices)	432
Selection-choices	7
Information Index (Selection-choices) ⓘ	77%
Imputation Ratio (Selection-choices) ⓘ	28.8
Existence Combinations (QOI DVs)	15
Discrete Imputation Ratio (QOI DVs)	1
Continuous Dimensions (QOI DVs)	9
Mean Active Continuous Dimensions (QOI DVs)	6
Continuous Imputation Ratio (QOI DVs)	1.5
Imputation Ratio (QOI DVs) ⓘ	1.5

Figure 3.24: ADORE graphical user interface, showing the architecture choices list.



Design Problem: Jet Engine SAO Problem

Design Variables

#	Name	Type	Source	Options	Fixed Value	Actions
1	Fulfill Generate Thrust	Discrete	Choice #3	2		
2	Fulfill Uncouple Fan RPM	Discrete	Choice #0	2		
3	Fulfill Exit Bypass Flow	Discrete	Choice #1	2		
4	Instantiate Component Compressor	Discrete	Choice #4	3		
5	Connect Port Bleed Air (1)	Discrete	Choice #7	3		
6	Connect Port Generator (1)	Discrete	Choice #8	3		
7	Design Variable (QOI) Splitter -> BPR	Continuous	QOI: BPR	Between 2 and 12.5		
8	Design Variable (QOI) Fan -> FPR	Continuous	QOI: FPR	Between 1.1 and 1.8		
9	Design Variable (QOI) Gearbox -> Gear Ratio	Continuous	QOI: Gear Ratio	Between 1 and 5		
10	Design Variable (QOI) Compressor 1/3 -> PR Param	Continuous	QOI: PR Param	Between 0 and 1		
11	Design Variable (QOI) Compressor 2/3 -> PR Param	Continuous	QOI: PR Param	Between 0 and 1		
12	Design Variable (QOI) Compressor 3/3 -> PR Param	Continuous	QOI: PR Param	Between 0 and 1		
13	Design Variable (QOI) Shaft 1/3 -> RPM	Continuous	QOI: RPM	Between 1000 and 20000		
14	Design Variable (QOI) Shaft 2/3 -> RPM	Continuous	QOI: RPM	Between 1000 and 20000		
				Between 1000 and		

Design Problem Settings

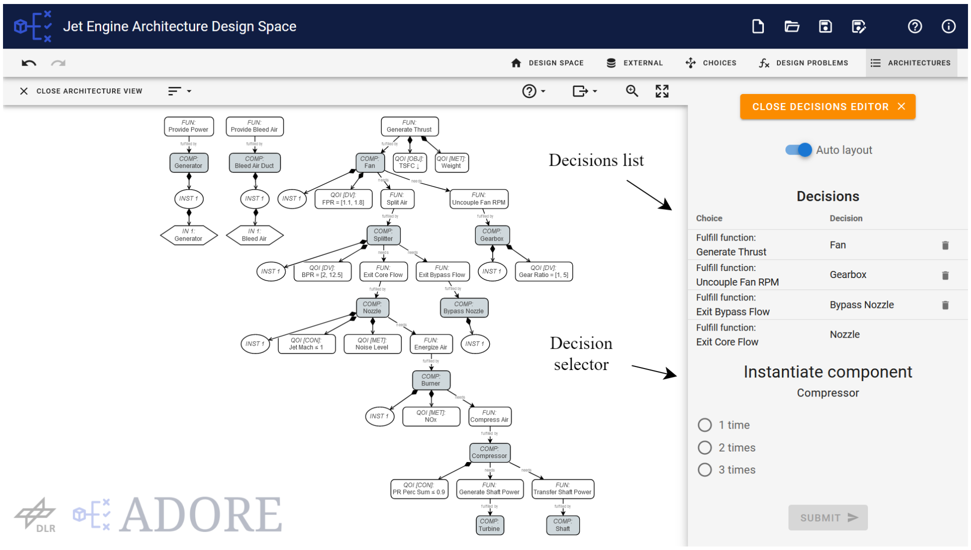
Jet Engine SAO Problem

Notes

Statistics

Design Variables	15
Fixed Design Variables	0
Free Design Variables	15
Discrete Design Variables	6
Continuous Design Variables	9
Valid Design Space (Discrete)	70
Declared Design Space (Discrete)	216
Discrete Choices	6
Discrete Imputation Ratio	3.1
Information Index	74%
Continuous Dimensions	9
Mean Active Continuous Dimensions	6

Figure 3.25: ADORE graphical user interface, showing a design (optimization) problem.



Decisions

Choice	Decision
Fulfill function: Generate Thrust	Fan
Fulfill function: Uncouple Fan RPM	Gearbox
Fulfill function: Exit Bypass Flow	Bypass Nozzle
Fulfill function: Exit Core Flow	Nozzle

Instantiate component

Compressor

- 1 time
- 2 times
- 3 times

Figure 3.26: ADORE graphical user interface, showing a manually-created partial architecture.

3.3.2. EVALUATION INTERFACES

Several interfaces for connecting to evaluation code have been developed as part of this work. Which interface is used depends on which analysis tools are available, in which computational environment, and how much programming skills the user has. Table 3.7 lists the available evaluation interfaces. The rest of this section discusses the interfaces in more details.

Table 3.7: Comparison of evaluation interfaces available in ADORE.

Environment	Interface	Paradigm	Usage Scenario
Python	Direct access	Imperative	Most flexible
	Class Factory Evaluator (CFE)	Declarative	Rule-based object creation
External	File-based serialization	Imperative	Serialization
	Node Factory Evaluator (NFE)	Declarative	Rule-based node creation

Direct Access The most flexible way to implement evaluation is by creating a class that subclasses ADORE's GRAPHAPIEVALUATOR class in Python, and to override the evaluation function: the function returning numerical values for all included output QOIs for a given architecture instance. The architecture instance is provided as an ARCHITECTURE object that provides details like the associated design vector and the included elements (subsystems, functions, components, component instances, connections, QOIs, etc.).

This *direct access* approach is the most flexible approach, however might require a steep learning curve and sufficient programming skills. Additionally, since the code then directly depends on the internal data model of ADORE, any changes to the data model (e.g. in future ADORE versions) might break such evaluation code.

Class Factory Evaluator Python-based evaluation can be supported by the *Class Factory Evaluator* (CFE). With the CFE, the user can define rules for instantiating objects based on selected architecture elements. To use the CFE, the user defines the following aspects (as Python code):

- *Class factories*: a class factory defines rules for instantiating a Python class into an object, consisting of an external element and class property definitions.

  The external element definition ("external" meaning external to the ADORE model) becomes part of an external database (see Section 3.3.1) and is linked to one or more ADORE elements by name matching (exact, using wildcards, or using regular expressions[2]).

  Class property definitions define how the values for the class properties are obtained: values can be static (e.g. string or numerical literal values), based on the linked ADORE element (e.g. the name, the ADORE data model object, or the component instance index), based on a port connection, or derived from a QOI.

[2] See for example https://regex101.com/ for an introduction to regular expressions.

- *Metric factories*: a metric factory defines an external element that is linked to an output QOI (i.e. an objective, constraint, or metric QOI).
- *The evaluation function*: the function that receives the ARCHITECTURE instance, instantiates objects using the class factories, uses these objects to run the problem-specific evaluation code, and then returns values for all the metric factories.

Compared to the direct access approach, the CFE approach reduces dependency on the internal data model of ADORE, and makes it easy to use evaluation code with object-oriented input.

File-based Serialization If analysis tools are not directly available in Python, it is also possible to use a *file-based* evaluation interface. Here, generated architectures are *serialized* as an XML, JSON, or RDF file, which are then used as input to an externally-integrated evaluation function. This approach is useful if another programming language is used to implement evaluation code, or for use in distributed evaluation approaches like (collaborative) MDAO integrated in dedicated Process Integration and Design Optimization (PIDO) platforms.

Node Factory Evaluator Analogous to the CFE for Python-based evaluation, the *Node Factory Evaluator* (NFE) can support file-based evaluation by defining rules on how to add nodes to some XML data file based on selected architecture elements. Compared to direct serialization, the NFE reduces dependency on the internal data model of ADORE, and may reduce the learning curve. The NFE is presented in more details in Section 4.2.

In computer science, a distinction is made between declarative and imperative programming approaches [253]. Declarative approaches describe the results to be obtained from some procedure, without prescribing the order in which functions and expressions need to be evaluated. Imperative approaches, however, describe every step to perform in order to obtain the desired results. The CFE and NFE interfaces can be regarded as declarative approaches, as here the user uses rules to define the desired result of the architecture conversion. Compared to this, the bare Python and file-based serialization interfaces are imperative approaches, as these require the implementation of additional code to interpret the architecture for evaluation. The declarative interfaces reduce the amount of code to be implemented to convert the architecture instances to the format needed for evaluation, compared to the imperative interfaces.

3.3.3. OPTIMIZATION INTERFACES

After the design space has been modeled and the evaluation function has been connected, the optimization problem can be formulated and executed. Running the optimization problem is done by connecting to SBArchOpt, presented in Section 2.5. ADORE provides an SBArchOpt problem definition, thereby allowing the use of any optimization algorithm in SBArchOpt. The problem can then be configured and executed directly using the Python APIs of SBArchOpt and ADORE.

3.4. THE FUNCTION-BASED ARCHITECTURE DESIGN SPACE MODELING PROCESS

This section provides a guideline for modeling the architecture design space using the ADSG as implemented in ADORE, and compares this approach to existing architecture design space modeling approaches. The modeling process is as follows:

1. Identify *boundary functions* from functional requirements. Boundary functions should be specified such that they are solution-neutral from the point of view of the system of interest [61].

   (Boundary) *functions* (FUN) should be formulated as "[*process*] [*operand*]" [39, 64], where process is a verb applied to an operand (e.g. material, energy, or signal flow). HIRTZ ET AL. [64] provide a useful database of solution-neutral processes and operands for functions of cyber-physical systems. Examples of functions are "accelerate air", "process signal", and "generate power".

2. For each unfulfilled function, define how they can be fulfilled:

   (a) Use a *concept* element (CON), mapping a (relatively) solution-neutral function to a (relatively) solution-specific function, to reduce the solution scope for the given design problem.
   Normally there are multiple concepts that can fulfill a given function. To then reduce the solution scope, one or more of the alternative concepts can be disabled. This documents that the architect is aware of other concepts that can be used to fulfill the given function, however for the given design problem these concept(s) are not considered.
   An example of this (shown in Figure 3.27) is an aircraft manufacturer that wants to design a new aircraft to fulfill the high-level function "transport passengers". Passengers can also be transported over land or water, which can be documented as alternative concepts, next to transporting by air. Transporting over land and water can then be disabled, because they will not be considered by the aircraft manufacturer.

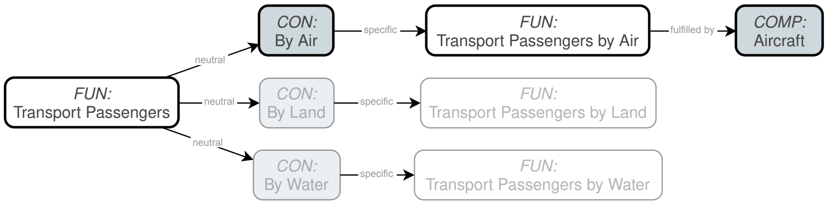

CON: By Air
specific
FUN: Transport Passengers by Air
fulfilled by
COMP: Aircraft
neutral
FUN: Transport Passengers
neutral
CON: By Land
specific
FUN: Transport Passengers by Land
neutral
CON: By Water
specific
FUN: Transport Passengers by Water

Figure 3.27: Example of function fulfillment by concepts, including disabled alternatives.

   (b) Use a *function decomposition* element (DE), mapping one function to multiple functions, if the following three conditions are satisfied:

   - "Parallelism" condition: satisfied if the function can be decomposed into two or more functions that either act in parallel, or are tightly-coupled in a loop without a clear starting point.

- "Solution-neutrality" condition: satisfied if the decomposed functions are as solution-neutral as the parent function, meaning they do not imply that some technology or solution is selected for fulfilling the parent function.
- "Compatibility" condition: satisfied if the fulfillment of the decomposed functions do not require major cross-tree incompatibility constraints to be defined later on.

For example, to control the flight path of an aircraft, it should be possible to control pitch, yaw, roll, and thrust (shown in Figure 3.28): the higher-level function "control flight path" can be decomposed into the lower-level functions "control pitch/yaw/roll/thrust". This mapping does not imply a specific control system solution, as all aircraft (including fixed-wing, rotorcraft, and lighter-than-air vehicles) need these functions. Also, they all act in parallel (there is no execution order between the four lower-level functions).

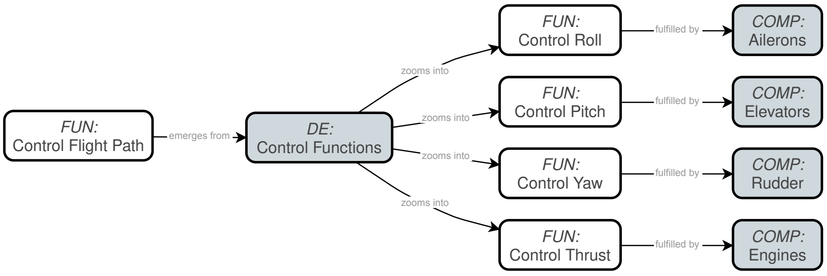


Figure 3.28: Example of function decomposition into parallel and solution-neutral functions.

(c) Use a *component* (COMP) if some element of form can fulfill the function. Components represent the implementation of the system, and components should be named as nouns. Components may induce additional functions, which in turn need to be fulfilled. Components can both fulfill and induce multiple functions. An example is shown in Figure 3.29.

Determining at what level of granularity to define components depends entirely on at what level architecture elements or architectural choices can be considered in the architecture evaluation code. For example, the function to "generate thrust" (for an aircraft) can be fulfilled by: a "propulsion system", a "turboprop", or a "propeller", in the order of increasing granularity. The "propeller" can be used to fulfill the function if there are alternatives for fulfilling the function, and/or if there are alternatives for fulfilling induced functions (e.g. electric motor or turboshaft for generating mechanical power), and if these alternatives can be captured in the evaluation code.

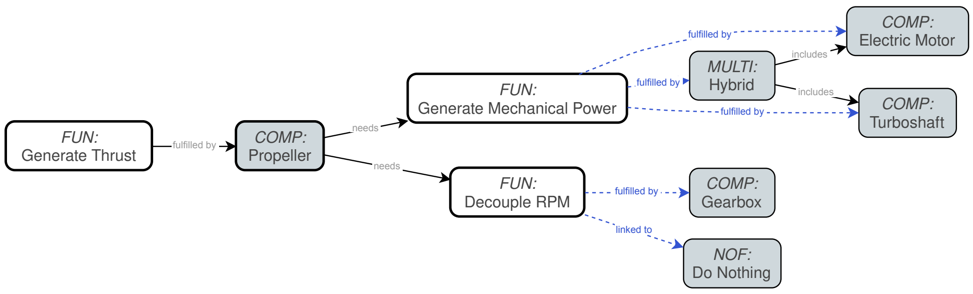


Figure 3.29: Example of function fulfillment by components, showing usage of multi-fulfillment and non-fulfillment elements. Architectural choices are shown by blue-dashed edges. Figure 3.18 shows the equivalent ADSG.

(d) Use a *multi-fulfillment* element (MULTI) if multiple components can fulfill a function simultaneously. For example, in a flight control system, the roll can be controlled by both ailerons and spoilers; in a hybrid-electric aircraft propulsion system, mechanical power can be provided to propellers by both electric motors and turboshafts.

Multi-fulfillment elements can be used to define selection choice options that act at the same level as components (and concepts and function decompositions). For example, if there are two alternative components to fulfill a function, the multi-fulfillment can be used to define the third option of using both components simultaneously. For an aircraft propulsion system this mechanism can for example be used to model the choice between full-electric propulsion (electric motor), conventional propulsion (turboshaft), and hybrid-electric propulsion (both), as also shown in Figure 3.29.

(e) Use a *non-fulfillment* element (NOF) to indicate the possibility for not fulfilling a function. This can for example be used for fulfilling secondary functions that may improve system performance, however at some cost. This indicates a trade-off which can be relevant at the system level.

An example would be the inclusion of a gearbox in a turbofan: the gearbox improves the propulsive efficiency of the turbofan, however at the cost of increased weight, manufacturing cost, and reduced reliability. This choice can be represented by a secondary function "decouple RPM", which can be fulfilled by a "gearbox" element or a non-fulfillment, as also shown in Figure 3.29.

3. Use *incompatibility constraints* to declare that two elements (functions or components) are not allowed to exist in an architecture instance together. This can be useful if the involved components are derived in different function paths or decomposition trees.

Incompatibility constraints are also useful if multiple components represent options in two or more architectural choices. If left unconstrained, each architectural choice is independent of each other and as such it can happen that more than one of the components is selected in an architecture, each for fulfilling different functions. If that does not represent a realistic architecture, then incompatibility constraints can be defined between the components.

An example is if for a flight control system, directional stability can be achieved by a vertical tail or by a V-tail, and longitudinal stability can be achieved by a horizontal tail or by a V-tail as well (shown in Figure 3.30). The combination of a V-tail and either a vertical or horizontal tail does not make sense, however if left unconstrained can be selected as an architecture instance.

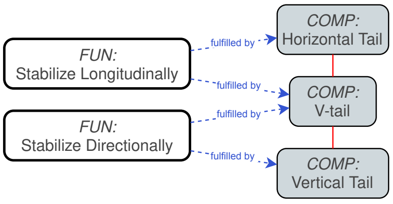


Figure 3.30: Example of incompatibility constraints (shown by red lines) to constrain independent choices.

4. Use *subsystems* (SYS) to group elements logically belonging together, especially if the group as such can be instantiated multiple times in an architecture.

   For example, elements inside of a propulsor unit can be grouped into a propulsor subsystem, if also other elements of the aircraft are modeled at a similar level of detail, or if the propulsor as a whole can instantiated multiple times. This example is shown in Figure 3.31, including a function fulfillment choice within the subsystem.

   Subsystems can be nested to enabling grouping elements at multiple levels of system decomposition.

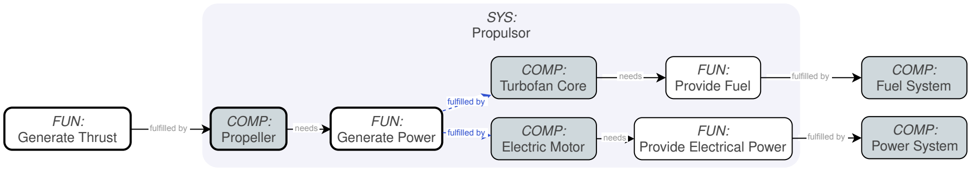


Figure 3.31: Example subsystem usage for logically grouping elements.

5. Once components have been defined, *function and component characterization* options can be defined:

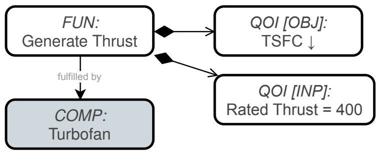
   - Component instantiation represents cases where the component can be included multiple times in an architecture, and where the number of instances is a choice. For example, the number of engines of an aircraft can be a choice.
   - Function and component static design variables represent sizing or property parameter choices, and can either be continuous, representing a number between two bounds, or discrete, representing a choice from a finite set

of options. Design variables can either be defined for the component as a whole, or for each component instance separately. Examples of design variables are the bypass ratio of a turbofan (a continuous design variable) or the number of compressor blade rows of a linear compressor (a discrete design variable).

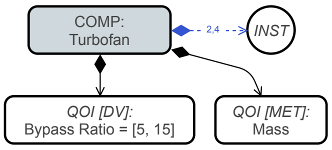
- Other characterization options include the definition of static inputs, for example representing assumptions or static requirements, and design constraints. Examples include defining environmental conditions as static inputs or defining material stress limit constraints.

Whether to use discrete design variables or alternative components for fulfilling a function depends on the level of granularity appropriate for the design problem. It is recommended to use alternative components if the different components induce different functions and/or contain different characterization options as described above. For example, a choice between two types of turbofans can be modeled using a turbofan component with a discrete design variable selecting between the different types . However, a choice between a turbofan, a turboprop, and other propulsion technologies might be more appropriately modeled using alternative components, as they might induce different functions to be fulfilled by other aircraft systems.

Static inputs, design variables, and design constraints are defined using Quantity of Interest (QOI) elements. QOI elements also define output metrics, optionally used as objectives or inequality constraints. Figure 3.32 shows an example of function and component characterization. The generate thrust function has two QOIs associated to it: a TSFC (Thrust-Specific Fuel Consumption) minimization objective and a rated thrust static input. The turbofan component has an instantiation choice (2 or 4 instances), and it has two QOIs associated: a bypass ratio continuous design variable and a mass output metric.



(a) System view showing an objective and a static input QOI associated to a function.



(b) Component view showing a design variable and a metric QOI associated to a component, and a component instantiation choice.

Figure 3.32: Example function and component characterization. Figure 3.19 shows the equivalent ADSG.

6. Architectural choices at the component-instance level or in instances of the same subsystem are independent of each other. If this should not be the case, the choices can be constrained by adding a *choice constraint*. The following types of choice constraints are available (see also Section 3.1.1):

    - "Linked": ensures that all choices are assigned the same option. This constraint can be applied if for example multiple propulsor subsystems exist in the example of Figure 3.31, and all subsystems should either have turbofans or electric motors.

- "Permutations": ensures that no choice is assigned the same option. The result is that the set of constrained choices effectively becomes a permutation choice.

  This can be useful if for example a selection of properties across component instances is fixed, however they may be distributed over components in different ways (i.e. permutations). It can also be used if there are many components for fulfilling some function across several subsystem instances, and each component can only be used in a specific subsystem once.
- "Unordered combinations": ensures that choices are assigned an equal or later option. This can be used to model situations where component-level or subsystem-level choices may have different options assigned over their instances, however where permutations (i.e. different ways of "ordering") of a set of options should not be considered.

  This can for example be used if in the example of Figure 3.31 it is allowed to have a mix of turbofans and electric motors across the different propulsor subsystems, and only the number of turbofans and electric motors influences the system performance (i.e. not the assigned subsystem index).
- "Unordered non-replacing combinations": ensures that choices are assigned a later option. The behavior is similar to the unordered combinations constraint, however prevents choices from having the same option assigned.

  This can be useful if there are many components for fulfilling some function across several subsystem instances, each component can only be used in a specific subsystem once, however permutations of technology assignment do not influence system performance.

7. Use *port* elements (PORT) to model connection, assignment, and/or allocation choices between (sets of) components.

   Such choices are appropriate when the components to be connected are included for fulfilling different functions (or as part of different function paths). For example, a turbofan might both have the primary function to generate thrust and a secondary function to generate electrical power. The primary function derives turbomachinery components and shafts with multiple compression stages. The secondary function might derive an electrical generator. The electrical generator is then connected to one of the shafts: which shaft instance to connect to can be modeled using a port element.

   Ports can also be useful to further elaborate how components within a function path are connected. For example, some power consumer (e.g. a flight control surface) might induce the function to provide power, which is fulfilled by a power distribution system. However, there are multiple control surfaces and multiple distribution systems for redundancy, and assigning the distribution systems to the control surfaces is not a trivial problem and is therefore relevant to be included in the architecture design space. This assignment choice can be modeled using a port, see Figure 3.33. Constraints can be added to for example ensure that each control surface is assigned to at least one distribution system, and each distribution system supplies at least one control surface with power.

Ports use connection choices (compared to selection choices for function fulfillments), see Section 3.1.2, which are more powerful for modeling architecture decision patterns as identified in [128]. It is for example possible to define connection, assignment, downselection, or permutation choices.

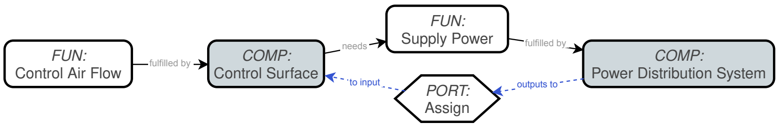


Figure 3.33: Example port usage to model an assignment choice. Figure 3.20 shows the equivalent ADSG.

8. Iterate until all functions are fulfilled, and elements are defined and characterized at the appropriate level of granularity.

   By modeling function fulfillment at the appropriate level of granularity, the "primary value paths" should emerge. A primary value path is the chain of functions that delivers the main value of the system (i.e. fulfills the boundary functions) by executing them sequentially or in parallel [39]. Note that primary value paths are properties of architecture instances, so architecture design spaces may contain multiple value paths.

9. Define system-level *performance metrics* to drive the architecture optimization loop. Performance metrics are defined using QOI elements (QOI), and can either take the role of minimization or maximization objectives, design constraints, or generic metrics:

   - Objectives are the main drivers of the optimization, and normally represent system-level metrics. They can be assigned to the appropriate boundary function, however they at least should be assigned to some element that is present in all architecture instances: otherwise architecture instances cannot be compared to each other.
   - Design constraints represent some threshold requirement to be met, and can either act at the system-level or at the component-level. Constraints do not have to necessarily always exist, for example if they constrain the behavior of a particular component only.
   - Generic metrics do not drive the optimization, however are useful for verifying evaluation results. They can for example represent internal calculation results (to verify that the final metrics are calculated correctly), or component-level calculation results (to verify that system-level aggregated metrics are calculated correctly).

   Figure 3.32 shows an example of QOIs used as objectives and output metrics.

10. Verify that the evaluation code is sensitive to the modeled architecture design choices: each choice should influence one or more performance metrics. If for a given choice this is not the case, there are two options:

    - remove the choice by disabling or removing associated model elements; or
    - (plan to) modify the evaluation code to be sensitive to the choice.

11. Verify that the architecture design space behavior is correctly implemented by verifying choices, adding choice constraints if needed, and manually generating architecture instances.

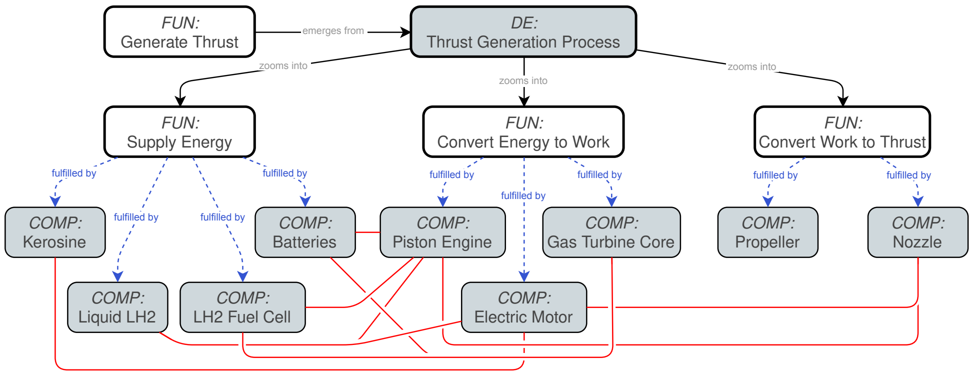
    The manual architecture generation process takes the architecture choice-by-choice from the design space model towards an architecture instance. By going choice-by-choice, the architect can verify that choices appear in a logical order, that the choice hierarchy is correctly implemented, that the correct options for different combinations of choices are present, and that any choice constraints are implemented correctly.

The consequence of using the function-based architecture design space modeling approach presented in this section is that the "zig-zagging" between functions and components happens along the primary value path. This approach therefore can also be seen as a *bottom-up* process, because the primary value path is defined step-by-step, starting from the final value-providing function (i.e. a boundary function). The presented bottom-up process is made possible by the fact that the DSG is a directed graph that may include cycles: this enables creating diverging derivation paths for function fulfillment, which can be merged and/or crossed-over further back along the primary value path. In contrast, tree-based approaches like feature models [69] or function-means trees [70] represent *top-down* processes, because the granularity appropriate for modeling function fulfillment choices for a given problem is mainly achieved using function decompositions. The resulting function fulfillment choices then reside in different branches of the tree, which requires the definition of cross-tree constraints to represent component (in)compatibilities.

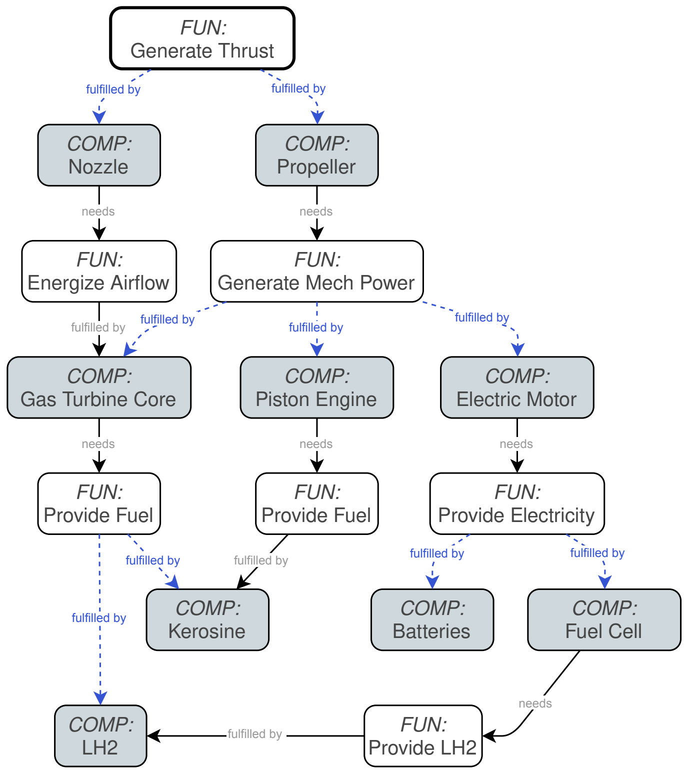
Figure 3.34 shows the architecture design space of an aircraft propulsion system, including several options for generating thrust, generating mechanical power, and supplying energy, modeled using both approaches. The top-down approach requires many cross-tree incompatibility constraints to ensure that only appropriate combinations of components can be selected. The bottom-up approach, however, requires no incompatibilities, because component compatibilities are defined using function fulfillment and induction relationships. Additionally, the top-down approach of the presented example violates two of the three conditions for using a function decomposition element:

- The parallelism condition is violated because there is a clear execution order between the decomposed functions (execution order: supply energy, convert energy to work, convert work to thrust), so there is no parallelism or tightly-coupled execution loop.
- The compatibility condition is violated because many cross-tree incompatibility constraints have to be defined in order to correctly model compatibilities.

The bottom-up process may thus result in less need for defining cross-tree constraints, prevents violating decomposition conditions, and therefore provides a more natural way to define function fulfillment choices.



(a) Top-down (tree-based) modeling approach.



(b) Bottom-up (graph-based) modeling approach.

Figure 3.34: Comparison between top-down and bottom-up architecture design space modeling approaches.

3.5. APPLICATION CASE II: HYBRID-ELECTRIC PROPULSION

This application case presents the design of a Hybrid-Electric Propulsion (HEP) system for a regional aircraft, to demonstrate the use of ADORE for defining and solving SAO problems. Hybrid-electric aircraft take part of their propulsion energy from conventional fuel sources such as kerosene, and part from electric sources such as batteries [254]. The architecture of the propulsion system plays an important role in the design of such aircraft, as it enables novel combinations of various electrical, mechanical and thermal components which must be investigated at the early stage of the design process due to their large impact on aircraft performance [255].

HEP Architecture Evaluation Framework A modular and flexible hybrid-electric propulsion system architecture builder and evaluation framework is used to enable dynamic formulation of the propulsion system evaluation model. This framework has been developed as part of this work, and is detailed in [24]. The evaluation framework is based on OpenConcept [256], an open-source mission analysis and propulsion system modeling framework. OpenConcept contains a set of component models and mission integration routines programmed using OpenMDAO [248], an open-source MDAO framework. The MDAO problem is setup automatically for each generated architecture by an architecture builder module. The architecture builder constructs an OpenMDAO model using OpenConcept library items from a definition of the propulsion system in a PROPSYSARCH class. Each architecture consists of thrust, mechanical power, and electrical power generation elements. Thrust generation elements convert mechanical power into thrust; mechanical power is generated from fuel or electrical power; and electrical power is generated from batteries and/or generators, and are only included if electrical power is needed for mechanical power generation.

In the OpenMDAO problem, the engines of the aircraft, the batteries required to complete the mission (if the architecture has batteries), propeller blade sizes, and airspeeds and vertical speeds along the mission are optimized by the objective:

$$f_{\text{sizing}}\left(\boldsymbol{x}_{\text{sizing}}\right) = (1 - t_{\text{coeff}})\,(w_{\text{fuel}} + 0.01\text{MTOW}) + 0.01\,t_{\text{coeff}}\,t_{\text{flight}}, \tag{3.2}$$

where t_{coeff} is an architecture-level design variable steering the optimization towards optimizing for flight time t_{flight} or towards fuel weight w_{fuel} and Maximum Take-off Weight (MTOW) optimization.

The aircraft is sized by the aircraft design tool OpenAD [257], which takes as inputs the top-level aircraft requirements (TLARs) and mission simulation results from OpenConcept. Aircraft requirements include the design range, cruise altitude, design payload, cruise Mach number and the wing loading. The output of OpenAD is a consistent aircraft design, including geometry, weights breakdown, and drag polar. These parameters are then input to OpenConcept, together with the mission definition, and PROPSYSARCH instance. OpenConcept then simulates the mission and calculates the propulsion system weight, amount of fuel used, and battery state-of-charge at the end of the mission. Fuel usages, the propulsion system weight, and mission segment durations are fed back into OpenAD. The cycle continues until OpenAD and OpenConcept results are consistent with each other: a good example of a coupled computational system as is common in MDAO [72].

The appropriateness of the architecture builder for SAO comes from the fact that all propulsion architectures have a common computational interface for connecting to analysis at a higher system level (e.g. mission analysis and aircraft sizing): the computational elements that are different between architecture instances are contained within the common interface, and are therefore effectively “hidden” from the rest of the computational problem. Additionally, the class-based definition used by the architecture builder is conveniently used with the Class Factory Evaluator (CFE; see Section 3.3.2). For example, a class factory might specify to instantiate the PROPELLER class if the propeller component occurs in an architecture instance, and assign properties taken from associated Quantities of Interest (QOIs) such as diameter and number of blades. Each component class that can be part of the PROPSYSARCH has an associated class factory, defining for which ADORE model elements it should be instantiated, and which QOIs from the ADORE model should be taken as property values.

The architecture evaluation function then comprises the following automated steps:

1. Instantiate architecture builder classes from the ADORE architecture instance using the CFE.
2. Assemble the PROPSYSARCH instance.
3. Build the OpenMDAO problem using the architecture builder.
4. Execute the OpenMDAO problem.
5. Extract performance metrics from the converged OpenMDAO problem.

Figure 3.35 displays the problem in XDSM [258] view, showing the interaction between the outer SAO loop and the inner aircraft sizing and mission analysis MDAO problem.

ADORE SAO Model The ADORE model representing the architecture design space is shown in Figure 3.36. The boundary function of the propulsion system is to Provide Propulsive Power [259]. This is fulfilled by the Thrust Generator component, which in order to produce thrust needs another function to be provided, which is to Accelerate Air. For this problem, only propellers are considered, although in a broader design space also turbofans or turboprops could fulfill this function. The optimization objectives, fuel consumption, MTOW, and flight duration are system-level QOIs, and are therefore associated to the boundary function. The Propeller induces two functions: Decouple RPM, which is fulfilled by a Gearbox, and Generate Shaft Power. Shaft power can be generated in three ways:

- by a Mechanical Turboshaft, which represents a conventional architecture;
- by an Electric Motor, which represents an electric architecture; or
- by a Mechanical Bus, which merges the Mechanical Turboshaft and Electric Motor power sources and therefore represents a hybrid-electric architecture.

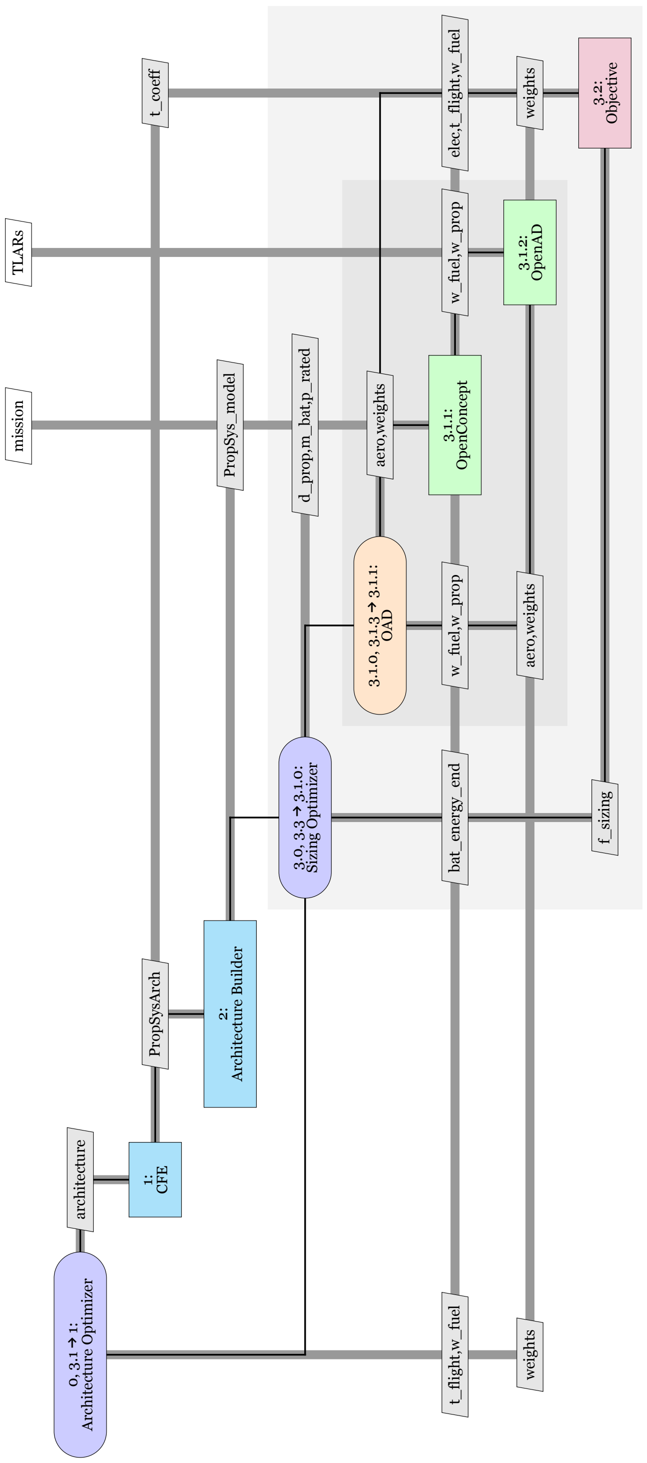


Figure 3.35: XDSM view of the hybrid-electric propulsion system problem, showing the interaction between the outer SAO loop and the inner aircraft sizing and mission analysis optimization loop.

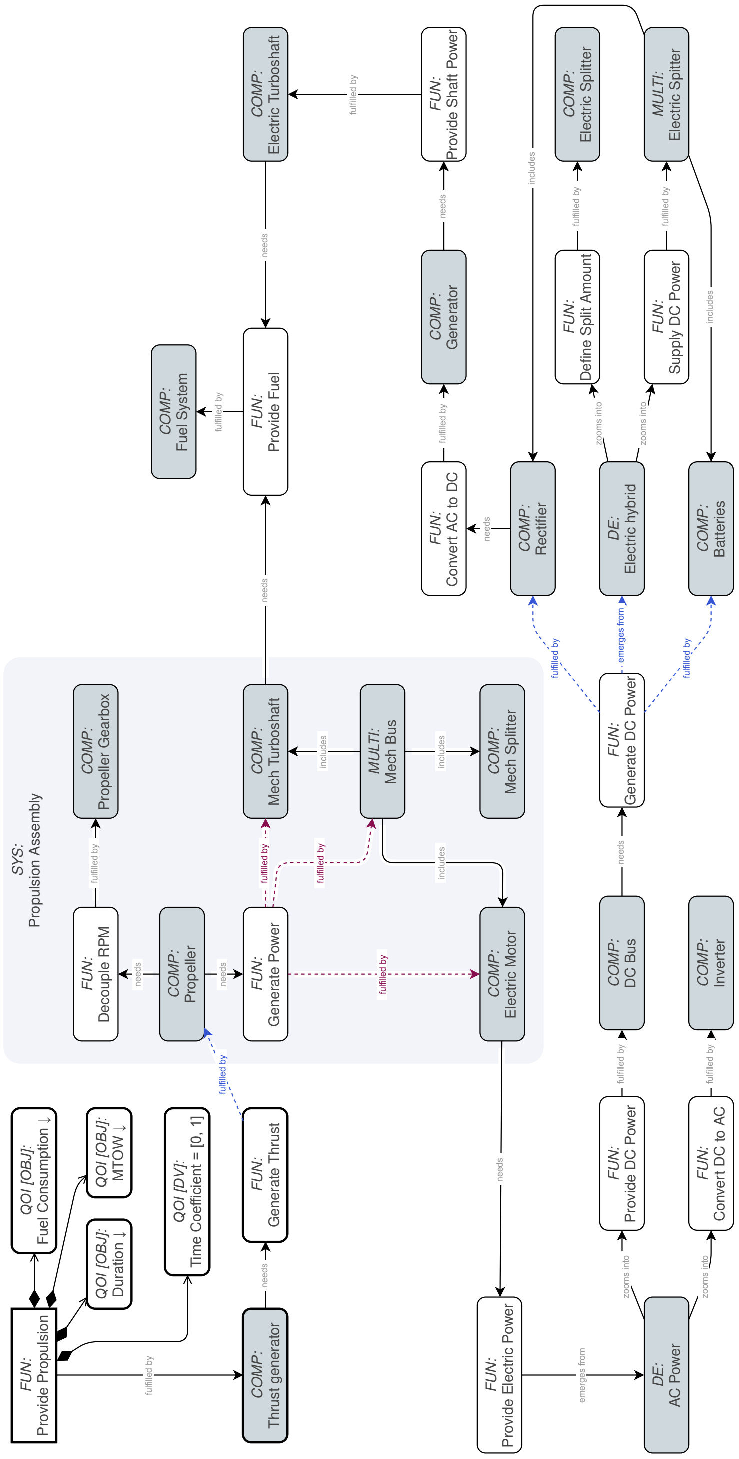


Figure 3.36: ADORE model of the hybrid-electric SAO problem.

If the Electric Motor is selected to provide (part of) the mechanical power, the architecture part related to electric power generation and distribution is included. Electric motors only work with AC power, and therefore Inverters are needed in order to convert the DC power provided by the DC bus into AC power. Then, there are three options for producing DC power:

- by Batteries;
- by a Generator that takes mechanical power from a Turboshaft and uses a Rectifier to convert AC power into DC power; or
- by a hybrid between the two, merged by an Electric Splitter[3] component.

Each propulsion assembly (propeller + gearbox + mechanical power source) is modeled as a subsystem, as this allows to define the number of assemblies as an architecture choice: between 1 and 5 propulsion assemblies per wing can be selected, and each assembly can have a different source of shaft power. However, the order at which the subsystems are placed along the wing has no influence on calculated performance metrics. For example, a wing with two propellers where the first has a turboprop and the second has an electric motor for mechanical power generation results in the same weight, fuel burn, and flight time as a wing where these two have swapped places. This limitation stems from the used analysis tools that cannot represent effects of engine order on for example wing root bending moment, aerodynamic interactions or other phenomena. In order to avoid repeated definition of architectures with the same output metrics, a choice constraint is applied such that only unordered combinations of mechanical power generation components can be defined. This constraint reduces the number of combinations of mechanical power source selections from $\sum_{k=1}^{5} 3^k = 363$ to $\sum_{k=1}^{5} \binom{3+k-1}{k} = 55$ [260], a reduction of 85%. The constrained nature of the choice is shown by purple-dashed lines in Figure 3.36.

[3]In OpenConcept, a "splitter" is a component for splitting one flow into multiple flows, but its behavior can easily be reversed so that it can be used as a merger instead.

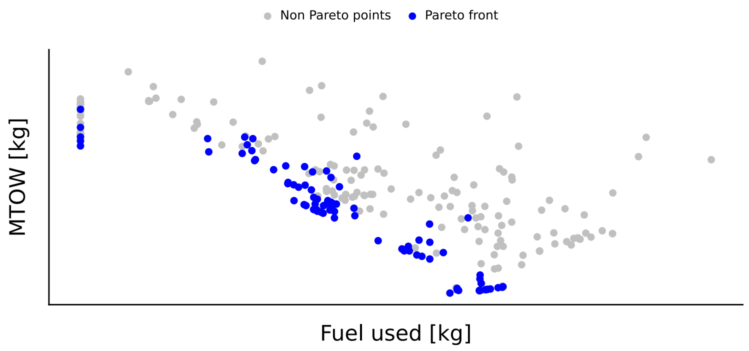
The optimization problem is encoded as:

$$
\begin{array}{lll}
\text{minimize} & w_{\text{fuel}}, \text{MTOW}, t_{\text{flight}} & \\
\text{w.r.t.} & 0 \leq t_{\text{coeff}} \leq 1 & \\
 & n_{\text{blades}} \in \{3,4\} & \\
 & n_{\text{prop}} \in \{1,2,3,4,5\} & \\
 & \text{GenPower}_i \in \{\text{Turboshaft}, \text{Electric}, \text{Hybrid}\} & i = 1,\dots,n_{\text{prop}} \\
 & \text{if GenPower}_i = \text{Hybrid}: & i = 1,\dots,n_{\text{prop}} \\
 & \quad 0 \leq \text{DOH}_{\text{mech},\text{<phase>},i} \leq 1 & i = 1,\dots,n_{\text{prop}} \\
 & \text{if any}\left(\text{GenPower} = \text{ElectricMotor} \vee \text{Hybrid}\right): & \\
 & \quad \text{GenDCPower} \in \{\text{Turboshaft}, \text{Batteries}, \text{Hybrid}\} & \\
 & \quad \text{if GenDCPower} = \text{Hybrid}: & \\
 & \quad\quad 0 \leq \text{DOH}_{\text{elec},\text{<phase>}} \leq 1 & \\
\text{subject to} & \text{idx}\left(\text{GenPower}_i\right) \leq \text{idx}\left(\text{GenPower}_{i+1}\right) & i = 1,\dots,n_{\text{prop}} - 1
\end{array}
$$

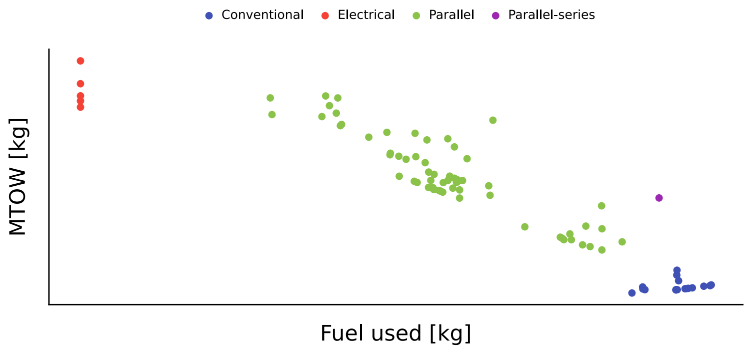
where DOH refers to the Degree of Hybridization, which is a parameter for determining what fraction of energy is generated by the electrified source compared to the conventional source. The DOH is defined separately for the mechanical ("mech" subscript) and electrical ("elec" subscript) power generation elements, and independently for each flight phase (denoted by the "<phase>" subscript): climb, cruise, descend, loiter, reserve climb, reserve cruise, and reserve descend. In total, the problem features 3 objectives, 4 constraints and 51 design variables, of which 43 are continuous and 8 are discrete (2 integer and 6 categorical). Additionally, all constraints and 47 of the variables are conditionally active.

Optimization Problem Execution The propulsion system architecture is designed and optimized for the King Air C90GT airframe: a twin-turboprop aircraft with a capacity for seven passengers. The SAO loop is driven by the Bayesian Optimization (BO) algorithm developed in Chapter 2, with an evaluation budget of 250 design points (Design of Experiments (DoE) size of 100, with 150 infill points). The default settings of the algorithm are used as described in Section 2.7: mixed-discrete hierarchical Gaussian Process (GP) models, constraint function mean value estimation as constraint handling technique, an ensemble of infill criteria, and a hierarchical sampling algorithm.

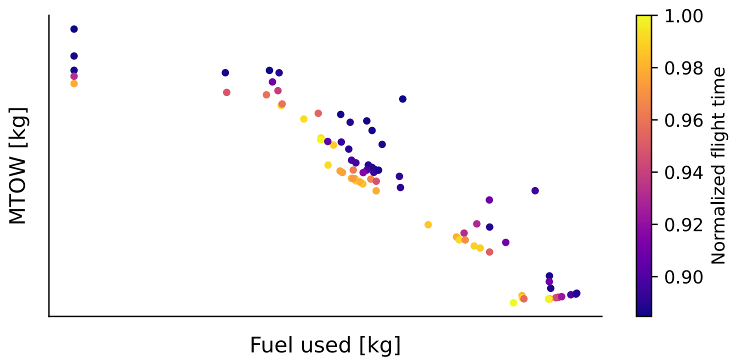
Figure 3.37 shows the optimization results. The Pareto front contains 88 design points. The extreme points of the Pareto front correspond to the fully electric architecture (lowest fuel consumption and highest MTOW) and the conventional architecture (highest fuel consumption and lowest MTOW). Between these two clusters of points, almost all the architectures that can be found in the Pareto front are parallel hybrid-electric architectures (i.e. mechanical power is generated both by electric motors and turboshafts). And in general, if a lower flight time is designed for, more energy will be necessary to complete the mission, in the form of a higher fuel consumption or a heavier aircraft.



(a) All evaluated design points, including the Pareto front in blue. Note that there are three objectives, however only two objectives can be plotted here.



(b) Pareto front showing architectures by power source.



(c) Pareto front showing normalized flight time.

Figure 3.37: Results of the HEP system application case.

3.6. Comparison to Manual SAO Problem Formulation

This section compares SAO problems as modeled and formalized using ADORE to manually-formulated SAO problems. The automated design variable encoding capabilities of ADORE are convenient for systems engineers, because then they do not have to be optimization experts in order to formalize the SAO problem. In order to be practically useful, however, the automated formulation should perform similarly or better than manually-formulated problems. Optimization performance is measured by ΔHV regret and calculated using the procedure described in Appendix E. 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. A lower ΔHV regret is better, as it shows that the optimum was approached more closely and/or in less iterations. Optimizations are performed using using NSGA-II, a multi-objective evolutionary algorithm (MOEA), and the BO algorithm developed in Chapter 2, using the default settings as described in Section 2.7.

The investigation is done with the three test problems from Section 2.3.1: the design of a multi-stage launch vehicle featuring selection choices (Section 3.6.1), the design of a guidance, navigation and control system featuring connection choices (Section 3.6.2), and the design of a jet engine featuring the Class Factory Evaluator (CFE, see Section 3.6.3). For each test problem, at least the following three formulations are compared:

- *Manual:* manual formulation.
- *ADORE Fast:* ADORE formulation, encoded using the fast selection choice encoder (see Section 3.1.1).
- *ADORE Complete:* ADORE formulation, encoded using the complete selection choice encoder (see Section 3.1.1).

The complete encoder uses $x_{\text{valid,discr}}$ to yield a more efficient encoding scheme in terms of imputation ratio (IR), correction ratio (CR), and therefore needs more memory and time for encoding compared to the fast encoder (which does not use $x_{\text{valid,discr}}$). However, the complete encoder uses design vector lookup for correction, and is therefore faster at correction than the fast encoder. To demonstrate these effects, next to optimization performance also the following aspects are measured:

- the hierarchical design space structure in terms of IR, CR, correction fraction (CRF), and max rate diversity (MRD), see Section 2.2 for their definitions; and
- the correction time (CT): the average time needed to correct and impute one design vector. The CT is relevant, because this operation may be called orders of magnitude more often than function evaluation, for example when generating a DoE or when searching for infill points in SBO.

3.6.1. Multi-Stage Launch Vehicle Architecture

The multi-stage launch vehicle SAO problem involves the choice of number of stages, several stage-level choices (number of engines, engine types, and stage length) and rocket geometry choices (head shape and length-to-diameter ratio). It is a multi-objective problem, with the goal to maximize payload mass (m_{payload}) and minimize cost (Cost) for a given target orbit altitude, subject to delta-V, structural and payload volume constraints (ΔV Margin, StructuralMargin and VolumeMargin, respectively).

SAO Problem Definition Figure 3.38 presents the ADORE model, showing the decomposition of the main function "Carry Payload to Orbit", the associated objectives, and the rocket stage subsystem. The Stage subsystem contains the rocket body component with the stage length design variable ($l_{\text{stage,factor}}$), and the engine assembly with the engine type choice (EngineType). Both the Stage and Engine Assembly subsystems can be instantiated 1, 2 or 3 times (n_{stages} and n_{engines}, respectively). Engine selection choices are constrained by a "linked" constraint, shown by the purple dashed edges, because it is not possible to select different engine types within a stage. The overall rocket geometry is further parameterized by the head shape selection (HeadShape) to fulfill the "Guide Airflow" function, which can either be a cone, elliptical or a semi-sphere. Selecting a cone additionally requires the selection of the cone angle (ConeAngle); selecting an elliptical head requires the selection of the length ratio (LengthRatio).

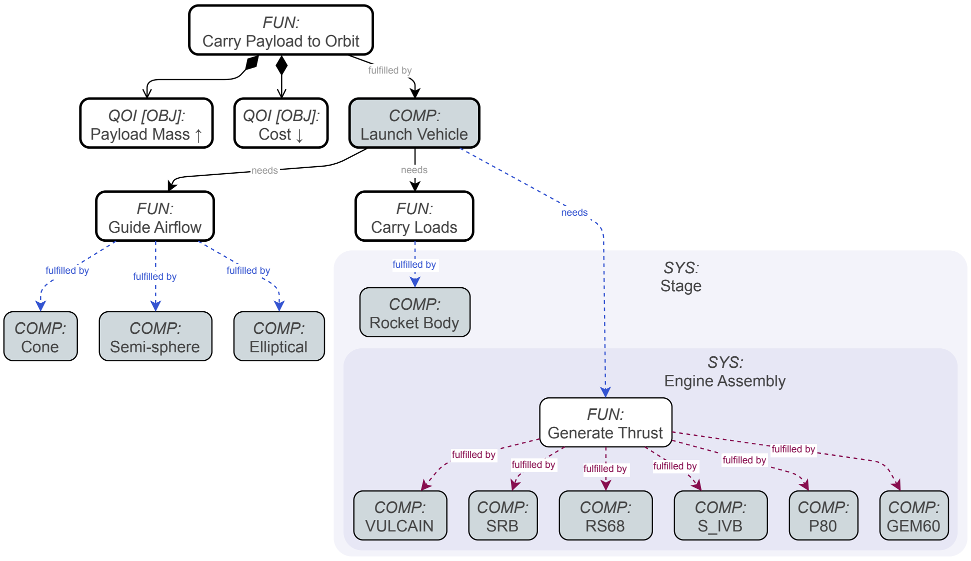


Figure 3.38: ADORE model showing the design space of the launch vehicle problem in system view.

The optimization problem is encoded as:

$$
\begin{array}{rll}
\text{minimize} & \text{Cost} & \\
\text{maximize} & m_{\text{payload}} & \\
\text{w.r.t.} & n_{\text{stages}} \in \{1,2,3\} & \\
 & n_{\text{engines},i} \in \{1,2,3\} & i = 1,\ldots,n_{\text{stages}} \\
 & \text{EngineType, i} \in & \\
 & \quad \{\text{SRB, P80, GEM60, VULCAIN, RS68, SIVB}\} & i = 1,\ldots,n_{\text{stages}} \\
 & 0 \leq l_{\text{stage,factor,i}} \leq 1 & i = 1,\ldots,n_{\text{stages}} \\
 & 10 \leq \text{LDRatio} \leq 11 & \\
 & \text{HeadShape} \in \{\text{Cone, Ellipse, SemiSphere}\} & \\
 & \text{if HeadShape} = \text{Cone}: & \\
 & \quad 28 \leq \text{ConeAngle} \leq 32 & \\
 & \text{if HeadShape} = \text{Ellipse}: & \\
 & \quad 0.15 \leq \text{LengthRatio} \leq 0.21 & \\
\text{subject to} & \Delta V\ \text{Margin} \leq 0 & \\
 & \text{StructuralMargin} \leq 0 & \\
 & \text{VolumeMargin} \leq 0 &
\end{array}
$$

In total, the problem features 14 design variables, of which 6 are continuous and 8 are discrete (4 integer and 4 categorical). Additionally, 8 variables are conditionally active.

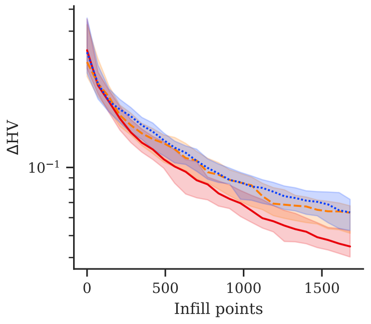
Comparison of Formulations Table 3.8 lists problem statistics. All formulations result in the same design variables and IR, CR, and CRF because of the low degree of hierarchical coupling: activeness is only determined by the number of stages and head shape design variables. Max rate diversity (MRD) also is the same for all formulations. MRD is relatively high, because 0.3% of valid discrete design vectors represent single stage rockets ($n_{\text{stages}} = 1$); 94% of valid discrete design vectors represents rockets with $n_{\text{stages}} = 3$. Correction time (CT) for the Manual and ADORE Complete formulations is significantly lower than ADORE Fast, because for the ADORE Fast formulation $x_{\text{valid,discr}}$ is not available and therefore a trial-and-error approach has to be used. ADORE Complete, however, still requires some model-parsing overhead, so Manual has the lowest CT. The manual formulation is available as LCRocketArch in SBArchOpt.

Table 3.8: Multi-stage launch vehicle problem formulations. Symbols and abbreviations: n_{x_d} = number of discrete design variables, n_{x_c} = number of continuous design variables, IR = imputation ratio, CR = correction ratio, CRF = correction fraction, MRD = max rate diversity, CT = correction time.

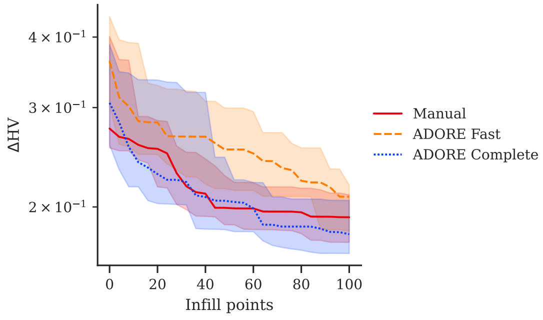
Formulation	n_{x_d}	n_{x_c}	IR	CR	CRF	MRD	CT [ms]
Manual	8	6	3.7	1.6	38%	94%	0.21
ADORE Fast	8	6	3.7	1.6	38%	94%	17
ADORE Complete	8	6	3.7	1.6	38%	94%	7.0
$n_{\text{valid,discr}}$	18 522						

Comparison of Optimization Performance The different formulations are solved using NSGA-II and the BO algorithm. Both algorithms start from an initial DoE of 70 points; NSGA-II is executed for 25 generations (population size 140) and 40 repetitions, BO is executed for 100 infill points with a batch infill size of 4, and 12 repetitions.

Figure 3.39 presents optimization results. For NSGA-II, the Manual formulation performs best and the two ADORE formulations perform similarly. The difference in performance is due to a different arrangement of design variables, leading to slight differences in the initial population. For the BO algorithm, the ADORE Fast formulation is performing worse than the Manual and ADORE Complete formulations.



(a) NSGA-II (40 repetitions)



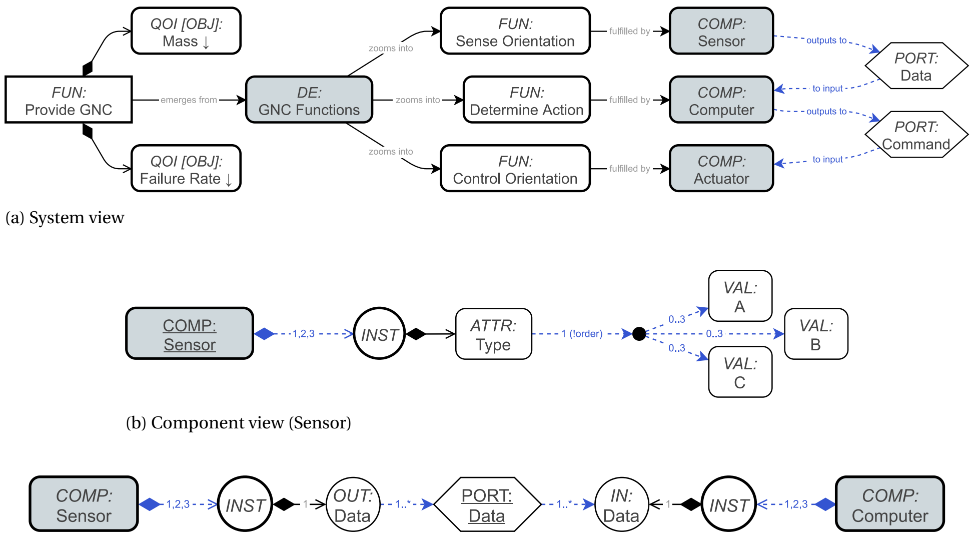
(b) BO (12 repetitions)

Figure 3.39: Multi-stage launch vehicle problem solved using two optimization algorithms. ΔHV represents the distance to the known Pareto front (Eq. (E.1)). The bands around the lines represent the 50 percentile range around the median.

3.6.2. GUIDANCE, NAVIGATION AND CONTROL ARCHITECTURE

The GNC (Guidance, Navigation & Control) problem [39, 122] features the definition of an architecture connecting sensors to flight computers, and flight computers to actuators. Each object (sensors, computers, actuators) can be instantiated 1, 2 or 3 times, and for each instance there are three types available with different masses and reliabilities associated to each (with in general increasing reliability as mass increases). The connections from sensors to computers and computers to actuators are architectural choices, with each connection increasing reliability of the system. A constraint is applied that ensures that no object is left unconnected. For each object, only unordered combinations of types can be assigned, as permutations of types lead to the same architecture if also the associated connections are permuted. The problem objectives are mass, calculated from the sum of selected object masses, and system-level reliability, calculated from a failure-tree approach assuming that the system does not fail as long as at least one sensor-computer-actuator path is still operational (i.e. the objects and connections have not failed). The interpretation of the problem is slightly different from [39, 122], so results cannot be compared directly.

SAO Problem Definition In ADORE, the architecture design space is modeled by decomposing the boundary function Provide GNC into Sense Orientation (fulfilled by Sensors), Determine Action (fulfilled by Computers), and Control Orientation (fulfilled by Actuators). Ports are used to model component connections: Data represents the sensor to computer connection, Command the computer to actuator connection. Figure 3.40a shows the system view, including the two system-level objective mass and failure rate, both to be minimized. Each component has 1, 2 or 3 instances, and an attribute specifying the type, see Figure 3.40b. Attributes are modeled as connection choices, "connecting" from attribute to value: in this case each component instance has an attribute needing exactly 1 connection, and each value can be connected to between 0 and 3 times. On the attribute side, order is set to irrelevant (shown by "!order"), effectively grouping outgoing connections and ensure only unordered combinations can be selected. Connections are modeled using ports, with the only constraint being that each connector has at least one connection (shown by "1..*"), see Figure 3.40c.



(a) System view

(b) Component view (Sensor)

(c) Port view (Data)

Figure 3.40: ADORE model showing the design space of the GNC problem in system view, component view and port view.

Component instantiation is modeled using selection choices, and these are relevant for determining all connector node existence scenarios for encoding connection choices. Therefore, it is expected that there will be a difference in behavior between the ADORE Fast and ADORE Complete formulations. The Manual formulation uses 3 integer variables for selecting the number of objects, 9 categorical variables for selecting object types, and 18 (2 times 9) binary variables for establishing object connections. The GNC problem features an additional formulation: "Encoded", which is manually formulated (i.e. not using ADORE), however uses connection choice encoders (see Section 3.1.2) for the connection choices. The Encoded formulation uses three combining pattern encoders (see also Table 3.4) for number of object selection (IR = 1, Dcorr = 100%), three unordered combining pattern encoders for object type selection (IR = 2.53, Dcorr = 16%), and two assigning pattern encoders for establishing connections (IR = 14.1, Dcorr = 100%). The Manual and Encoded formulations are available in SBArchOpt as GNC and AssignmentGNC, respectively. The ADORE versions of the optimization problem are encoded as:

$$\begin{array}{lll}
\text{minimize} & \text{Mass}, \text{FailureRate} & \\
\text{w.r.t.} & n_{\text{sensors}} \in \{1,2,3\} & \\
& n_{\text{computers}} \in \{1,2,3\} & \\
& n_{\text{actuators}} \in \{1,2,3\} & \\
& \text{type}_{\text{sensor,i}} \in \{\text{A},\text{B},\text{C}\} & i = 1,\ldots,n_{\text{sensors}} \\
& \text{type}_{\text{computer,i}} \in \{\text{A},\text{B},\text{C}\} & i = 1,\ldots,n_{\text{computers}} \\
& \text{type}_{\text{actuator,i}} \in \{\text{A},\text{B},\text{C}\} & i = 1,\ldots,n_{\text{actuators}} \\
& \text{connect}_{\text{sensor}\rightarrow\text{computer},i} \in \{0,1\} & i = 1,\ldots,n_{x,\text{connect}} \\
& \text{connect}_{\text{computer}\rightarrow\text{actuator},i} \in \{0,1\} & i = 1,\ldots,n_{x,\text{connect}}
\end{array}$$

The problem only features discrete variables, of which 3 are integer (the variables selecting the numbers of elements). The element type selection variables are constrained to ensure only unordered combinations are selected, as discussed before. The connection variables are encoded using connection choice encoding algorithms (see Section 3.1.2), and therefore the number of variables depends on the specific problem formulation (yielding $n_{x,\text{connect}}$ categorical variables) and cannot be determined statically.

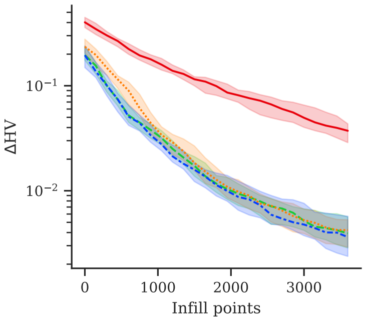
Comparison of Formulations Table 3.9 presents statistics of the GNC problem, including and excluding actuators. It shows that the ADORE Complete formulation obtains the same problem definition as the Encoded formulation. The Manual formulation results in a higher IR and CR compared to the Encoded and ADORE Complete formulations, showing that automatically choosing the connection choice encoders improves problem formulation. ADORE Fast formulation results in a very high IR, because the fast selection choice encoder is not able to correctly determine all connector node existence scenarios, which results in less efficient connection choice encoding. MRD is the same for all formulations. Manual formulation has the fastest correction time (CT) as it features problem-specific code tailored to the formulation. Encoded and ADORE Complete formulations are slightly slower, as they depend on greedy correction that uses design vector lookup. ADORE Fast correction is slowest as it depends on trial-and-error correction.

Comparison of Optimization Performance The different formulations (including actuators) are solved using NSGA-II and the BO algorithm. Both algorithms start from an initial DoE of 150 points; NSGA-II is executed for 25 generations (population size 150) and 40 repetitions, BO is executed for 100 infill points with a batch infill size of 4, and 12 repetitions. Figure 3.41 presents optimization results. It shows that for NSGA-II, the Manual formulation performs significantly worse than the other formulations, which all perform similarly. For BO, all formulations perform similarly, with the Manual formulation performing slightly worse than the others. It can be concluded that the

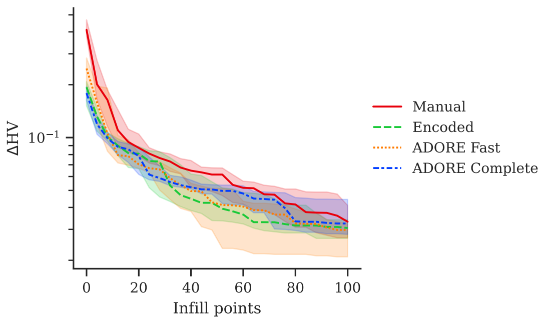
Table 3.9: GNC problem formulations. The "Encoded" formulation is manually formulated (i.e. not using ADORE), however uses the connection choice encoders from Section 3.1.2. Symbols and abbreviations: n_{x_d} = number of discrete design variables, IR = imputation ratio, CR = correction ratio, CRF = correction fraction, MRD = max rate diversity, CT = correction time.

	Including actuators			Excluding actuators					
Formulation	n_{x_d}	IR	CT [ms]	n_{x_d}	IR	CR	CRF	MRD	CT [ms]
Manual	30	1761	4.7	17	113	17.2	60%	94%	1.4
Encoded	33	367	38	19	39.5	6.0	49%	94%	12
ADORE Fast	30	23460	650	17	632	—	—	94%	102
ADORE Complete	33	367	62	19	39.5	6.0	49%	94%	17
$n_{\text{valid,discr}}$		79 091 323				29 857			

ADORE formulations all result in problem formulations that can be solved by EA and BO algorithms, and that they perform as well as the Encoded formulation and better than the Manual formulation.



(a) NSGA-II (40 repetitions)



(b) BO (12 repetitions)

Figure 3.41: GNC problem (including actuators) solved using two optimization algorithms. ΔHV represents the distance to the known Pareto front (Eq. (E.1)). The bands around the lines represent the 50 percentile range around the median.

3.6.3. JET ENGINE ARCHITECTURE

The jet engine architecture problem features the selection and sizing of jet engine components. The same problem definition as in Section 2.6 is used, which is a single-objective problem minimizing Thrust-Specific Fuel Consumption (TSFC) subject to several feasibility constraints. Architectural choices include adding a fan (turbofan architecture) or not (turbojet architecture), the number of compressor/turbine stages, whether bypass and core flows are mixed before flowing out, whether a gearbox is added between the fan and low-pressure shaft, and the locations of bleed air and power offtakes. Continuous sizing variables include bypass ratio, fan pressure ratio, compressor pressure ratios, gearbox ratio and shaft RPM's. Refer to Section 2.6 for more details.

SAO Problem Definition Figure 3.42 shows the ADORE model with three boundary functions: Generate Thrust, Provide Bleed Air, and Provide Power. The Generate Thrust function has the TSFC objective and weight metrics associated to it, and its fulfillment represents the choice whether to include a fan or not. The nozzle mixing choice is represented by the fulfillment of the Exit Core/Bypass Flow functions. Incompatibility constraints are used to model the constraint that either a mixed nozzle is selected, or both the (core) nozzle and bypass nozzle. The (core) nozzle and mixed nozzle both need the Energize Air function, which derives the remaining components in the engine core. The Compressor, Shaft and Turbine components include instantiation choices (1, 2 or 3), which are linked by a choice constraint ensuring that the number of instances match. The Provide Bleed Air and Power functions are fulfilled by connecting a Bleed Air Duct and Generator to one of the Compressors and Shafts, respectively. The gearbox choice is an example of non-fulfillment: the function Uncouple Fan RPM represents an "improvement" function which can either be fulfilled by the gearbox, or it can be left unfulfilled. Sizing variables (e.g. bypass ratio, compressor ratios) are modeled as design variable QOIs of the respective components.

Jet engine architecture instances are sized and evaluated using the framework presented in Section 2.6, which constructs an MDAO problem using pyCycle [249] and OpenMDAO [248] based on a set of ARCHELEMENT classes that represent engine elements. In the ADORE formulations, ARCHELEMENT classes are instantiated from architecture elements using the Class Factory Evaluator (CFE) interface. The Manual formulation of the problem is available in SBArchOpt as SIMPLETURBOFANARCH.

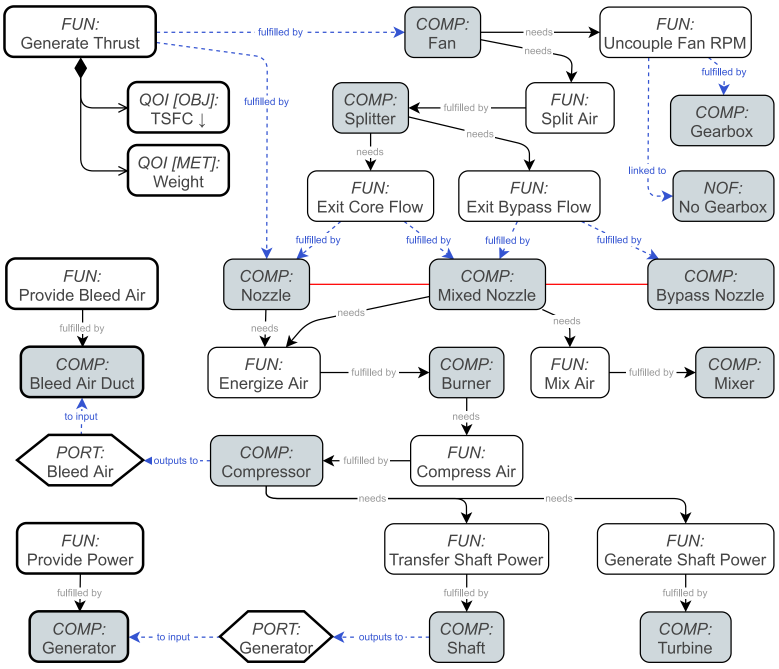


Figure 3.42: ADORE model showing the design space of the jet engine problem in system view.

Comparison of Formulations Table 3.10 lists statistics, showing that the Manual and ADORE Complete formulations are similar, and the ADORE Fast formulation leads to a higher Imputation Ratio (IR). MRD is the same for all formulations. Manual and ADORE Complete CT values are significantly lower than ADORE Fast due to $x_{\text{valid,discr}}$ availability.

Table 3.10: Jet engine problem formulations. Symbols and abbreviations: n_{x_d} = number of discrete design variables, n_{x_c} = number of continuous design variables, IR = imputation ratio, CR = correction ratio, CRF = correction fraction, MRD = max rate diversity, CT = correction time.

Formulation	n_{x_d}	n_{x_c}	IR	CR	CRF	MRD	CT [ms]
Manual	6	9	3.9	2.1	55%	60%	9.0
ADORE Fast	7	9	9.3	2.1	33%	60%	87
ADORE Complete	6	9	4.6	2.1	55%	60%	17
$n_{\text{valid,discr}}$					70		

Comparison of Optimization Performance Optimizer performance is compared for the BO algorithm. The algorithm is executed with an initial DoE of 75 points, 200 infill points, 4 points evaluated in parallel, and 12 repetitions. NSGA-II is not used for this SAO problem, because evaluating one design vector takes between 1 and 5 minutes. Figure 3.43 presents optimization results, showing that the Manual and ADORE Complete formulations perform similarly, and ADORE Fast performs slightly worse.

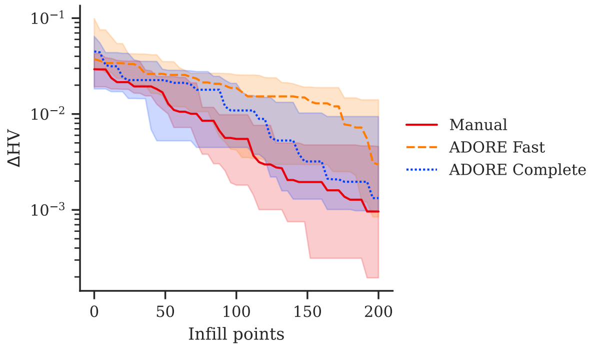


Figure 3.43: Jet engine problem solved using the BO algorithm (12 repetitions). ΔHV represents the distance to the known Pareto front (Eq. (E.1)). The bands around the lines represent the 50 percentile range around the median.

3.7. Chapter Conclusions

In this chapter the architecture design space modeling approach developed in this research has been presented. The tasks of the modeling approach as integrated in the SAO loop are to define architecture design choices (architectural choices and architecture-specific design parameters), encode them as design variables, convert design vectors to architecture instances, and to interpret performance metrics as objectives and constraints. The developed design space modeling approach is built-up of three layers:

- The Design Space Graph (DSG; Section 3.1) implements the mechanism for modeling hierarchical selection and connection choices using a directed graph. The design space can be constrained using incompatibility constraints and choice constraints. The problem formulation is completed by the definition of additional design variables and metrics that can take the role of objectives or constraints. Choices are automatically encoded using a complete or fast encoder for selection choices and a repository of encoding algorithms for connection choices. The implementation of the DSG is available open-source.
- The Architecture Design Space Graph (ADSG; Section 3.2) adds a semantic layer on top of the DSG to use it in a function-based system architecting context. Various node types representing system architecture concepts are defined, such as functions, components, component instances, and ports. Rules are defined for which nodes can be connected to which nodes, and where selection choices are automatically inserted.
- ADORE (Section 3.3) is the tool for editing the ADSG in a web-based GUI. It defines a user-friendly model visualization with various views, and several additional elements like subsystems and Quantities of Interest (QOIs). ADORE also provides interfaces for connecting to evaluation code, either directly using the Python API or through file-based interfaces. Finally, it also provides a problem definition interface to SBArchOpt (see Section 2.5) for solving SAO problems.

Section 3.4 presents the bottom-up process for modeling architecture design spaces using ADORE and the ADSG. Guidelines are defined for defining functions, fulfilling functions using components, multi-fulfillment or non-fulfillment elements, managing complexity using decompositions, concepts, and subsystems, defining component characterization and connection choices, for using incompatibility and choice constraints, and for defining performance metrics. It is shown that the bottom-up process results in a more convenient design space definition compared to top-down approaches.

The modeling approach is then demonstrated by a hybrid-electric propulsion (HEP) system architecture optimization problem (Section 3.5), showing:

- an object-oriented way for implementing a locally-executed, dynamically formulated MDAO problem used as architecture evaluation function;
- the use of the Class Factory Evaluator (CFE) to instantiate PropSysArch classes based on selected architecture elements;
- an ADORE model defining the function-based architecture design space, featuring QOI definitions, choice hierarchy, a subsystem, and a choice constraint; and

- the execution of the three-objective SAO problem using the Bayesian Optimization (BO) algorithm developed in Chapter 2, resulting in a Pareto front of optimal HEP architectures.

Finally, Section 3.6 shows that SAO problems modeled using ADORE and encoded by the ADSG result in formulations that perform similarly or better than manually-encoded SAO problems, determined by how quickly and closely the optimum is approached (measured by ΔHV regret). The complete selection choice encoder performs better than the fast selection choice encoder. This results shows that ADORE can be used to define and solve SAO problems, and that the system architect thus does not need to be an expert in numerical optimization to manually define the design variables.

With these results, the second sub-objective has been achieved:

2. Develop a way to formulate SAO problems, consisting of:
   (a) a method for modeling SAO problems based on system functions, that supports the definition of architectural choices and architecture-specific design parameters; and
   (b) algorithms to encode the model as an optimization problem (i.e. in terms of design variables, objectives, and constraints), and to decode design vectors into architecture instances.

4 Collaborative Multidisciplinary Evaluation

Chapter Contents

Multidisciplinary Design Analysis and Optimization (MDAO) enables coupling analysis tools stemming from various engineering disciplines, none of which can be ignored or should be allowed to take the overhand in defining the design, into one coupled computational system to analyze and optimize the product at the system-level [72]. The utility of MDAO for System Architecture Optimization (SAO) has already been demonstrated for the design of a hybrid-electric propulsion system in Section 3.5. There, the automatically-constructed MDAO problem that couples mission analysis to overall aircraft design was used to get a system-level estimate of weight, energy usage, and flight time for a given system architecture. That MDAO system, however, was locally integrated and executed, something which would be difficult or impossible if a more distributed and diverse set of disciplinary tools is needed to design a system.

Collaborative MDAO allows coupling diverse disciplinary analysis tools, developed and managed by different teams potentially from different organizations, in a single MDAO workflow [36]. It is characterized by the application of a Central Data Schema

This chapter is partly based on [2, 8, 9].

(CDS) and the distributed, cross-organizational management, development, and execution of analysis tools. This chapter presents the developments done to enable the application of collaborative MDAO for SAO, according to the three requirements placed on evaluation functions as defined in Section 1.5.5:

- The evaluation function should be flexible enough to handle all architectures. For collaborative MDAO this means that the workflow should *automatically* modify its execution behavior (i.e. data connections and tool inclusion and execution order) according to the architecture instance being evaluated.

  Section 4.1 presents high-level strategies, identifies the influences that changing the system architecture might have on the workflow behavior, and shows how this can be implemented in a collaborative MDAO modeling tool.
- The evaluation function should be sensitive to relevant architecture design choices. For collaborative MDAO this means that *all relevant information* about the architecture instance (e.g. function allocation, component selection and characterization, component connections) should be communicated to the MDAO workflow through its Central Data Schema (CDS).

  Section 4.2 presents a mechanism for translating an architecture instance into the CDS of a collaborative MDAO workflow.
- The evaluation function should be executed without user interaction. For collaborative MDAO this means that the architecture generator and MDAO workflow should be executed in a *shared computational environment*.

  Section 4.3 presents a strategy for integrating the architecture generator in the Process Integration and Design Optimization (PIDO) environment where the collaborative MDAO workflow is deployed.

The design of a multi-stage launch vehicle demonstrates the combination of these three aspects in Section 4.4. Section 4.5 concludes the chapter.

4.1. Dynamic Behavior in MDAO Workflows

This section discusses how MDAO problems can be made flexible enough to calculate system performance for all architecture instances in a given SAO problem. First, how architectural changes influence MDAO workflow behavior is discussed in Section 4.1.1. Then, Section 4.1.2 presents high-level strategies for implementing such behavioral changes in MDAO. Finally, an implementation of one of the high-level strategies to support dynamic collaborative MDAO is presented in Section 4.1.3.

4.1.1. Architectural Influences on MDAO Workflow Behavior

Traditionally, MDAO has been used to formulate and solve computational problems only involving continuous variables [210]. However, some research has been performed into also supporting discrete state and coupling variables in MDAO, as would be needed for SAO [49]. The following discussions extend this state-of-the-art to the support for dynamic behavior of MDAO workflows for SAO.

MDAO problems are formulated and solved for a given system architecture, and changing the system architecture requires changing the MDAO formulation by adding or removing disciplines, or partly redefining discipline inputs or outputs [36]. The mechanisms by which a change in system architecture may trigger a change in MDAO problem formulations are called *architectural influences*. Four architectural influences have been identified as part of this work:

1. conditional variables;
2. data connection;
3. discipline repetition; and
4. discipline activation.

The following subsections introduce the influences, as well as proposed modifications of the Extended Design Structure Matrix (XDSM) [258] notation to show them. For more background on how the four architectural influences were derived as part of this work, the reader is referred to [261].

Influences on Variable Routing

Two influences impact the routing of variables: "conditional variables" and "data connections".

Conditional Variables System components are parameterized by variables in the CDS, therefore adding or exchanging components leads to the addition or removal of variables. Such variables are called *conditional variables*, because they may exist depending on architectural choices. Conditional variables are similar to conditionally active design variables (see Section 1.5.3), however can also represent state and coupling variables in the MDAO problem.

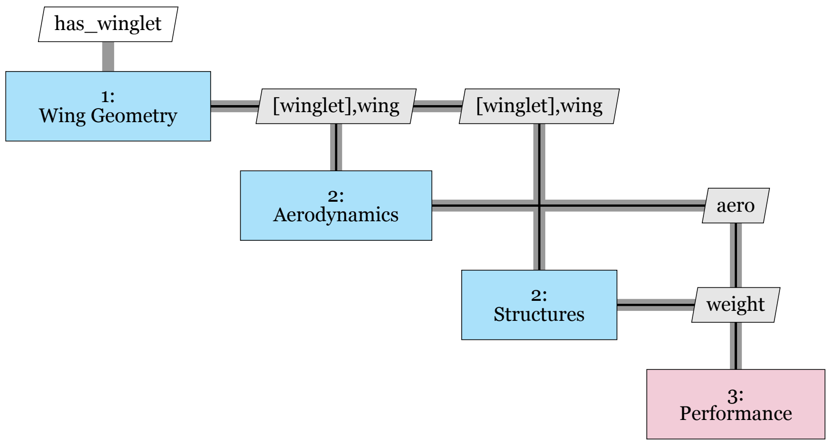


Figure 4.1: Example of a "conditional variables" influence: only if the wing has a winglet, the winglet variables (e.g. angle, length, taper ratio) are passed to the aerodynamics and structures tools. Conditional variables are wrapped by square brackets.

An example of this could be an architectural choice about whether or not to include a winglet on a wing, see Figure 4.1: if included, new state and/or coupling variables such as angle, length and taper ratio will be defined in the CDS, along with their usage in the MDAO problem, for example for aerodynamic and structural analysis. In the XDSM, conditional variables are wrapped by square brackets (see Figure 4.1).

Data Connections A coupling variable represents an output from a tool and an input to one or more other tools: the coupling variable represents a data connection. If due to some architectural choice, the tool from which the variable is an output or the tool to which the variable is an input changes, this is called a *data connection* influence. In the case of this influence, the variable itself always exists (so it is not a conditional variable), however its usage in terms of discipline input and/or output may change.

An example of this influence could be a choice about landing gear attachment, see Figure 4.2: whether the landing gear is attached to the wing or to the fuselage determines whether landing gear loads (for example output by a mass or inertia discipline) are inputs to the fuselage or wing structural sizing discipline. In the XDSM, variables subject to a data connection influence are displayed in bold-italic font (see Figure 4.2). The logic determining when which data connection is active is not displayed, in order to save space.

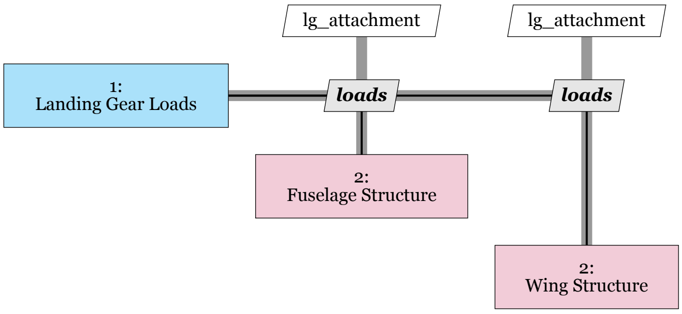


Figure 4.2: Example of a "data connection" influence: the Loads will be transferred to the Fuselage Structure tool if the landing gear is attached to the fuselage, and to the Wing Structure tool if the landing gear is attached to the wing. Variables that are subject to data connection influences are shown in bold-italic.

Influences on Tool Execution

Two influences are related to disciplinary tool execution: discipline repetition and discipline activation.

Discipline Repetition When disciplinary tools are directly mapped to system components, architectural choices may influence usage of these disciplinary tools. If the number of instances of a component is an architectural choice, *discipline repetition* may be needed. Discipline repetition involves variations in the inputs or outputs, as each discipline iteration uses and produces slightly different sets of inputs/outputs. Note that discipline repetition only refers to in-place repetition of tool execution, not to including a tool multiple times in the workflow at different locations.

An example of a discipline repetition influence could be a disciplinary tool that simulates one aircraft engine at a time, see Figure 4.3: the number of times the discipline should be repeated in the MDAO workflow then depends on the number of engine instances in the system architecture. In the XDSM, a repeated discipline is shown using the stacked representation, and by showing the variable that determines the number of repetitions in the top-right corner of the block, after an "x" meaning "times" (see Figure 4.3). A subscript "i" appended to variable names indicates that each repetition uses/provides a different index of that variable.

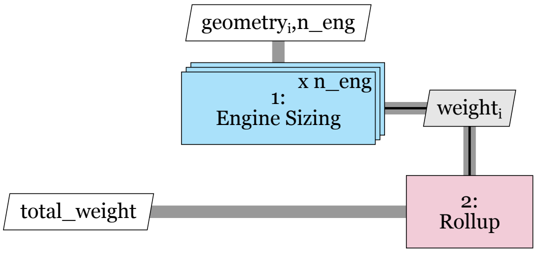


Figure 4.3: Example of a "discipline repetition" influence: the propulsion performance tool is executed once for each engine instance, determined by the number of engines "n_eng". A repeated discipline is shown using the stacked representation used for parallel execution in the original XDSM notation [258], and by showing the variable determining the number of repetitions, prefixed by an "x" (meaning "times"), in the top-right corner of the block. Geometry and weight are variables of which each repetition takes/provides a different index, as indicated by the subscript "i".

Discipline Activation The other architectural influence that determines tool usage is *discipline activation*. This influence simply determines whether a discipline is used or not based on an architectural choice.

Figure 4.4 shows an example of the activation of a tool that sizes the structure for composite materials and simultaneous deactivation of a tool that does so for metallic materials, based on the choice of material for some part. In the XDSM, the activation condition of a tool subject to activation is shown below the tool name, after "Activation:" (see Figure 4.4). BRUGGEMAN ET AL. [213] instead visualize tool activation by using a switch element, which output decision-variables which determine which one of several branches (a branch can be a tool or a group of tools) is executed.

Any combination of influences may exist in an MDAO workflow, and some influences may trigger other influences. For example, discipline activation or repetition may also involve changes in input or output definitions, which may make some variables conditional or result in data connection changes. Influences may apply to groups of disciplines as well, as also observed by BRUGGEMAN ET AL. [213]. This is in agreement with [19], where it is observed that groups of disciplines may be collapsed into a block with the union of the inputs and outputs while preserving the MDAO workflow execution logic.

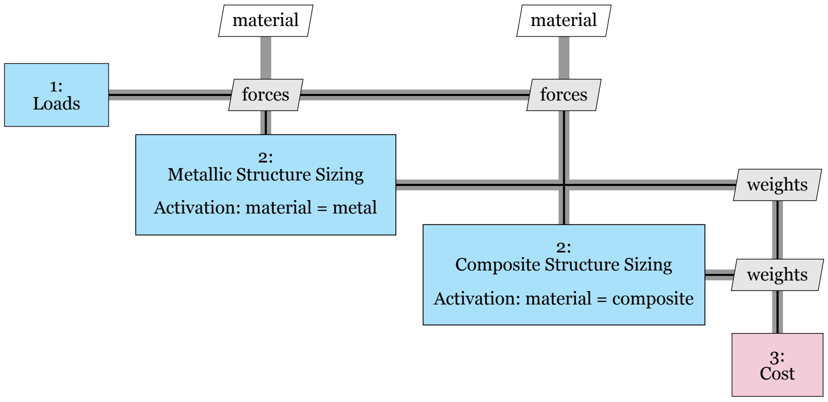

material
material
1: Loads
forces
forces
2: Metallic Structure Sizing
Activation: material = metal
weights
2: Composite Structure Sizing
Activation: material = composite
weights
3: Cost

Figure 4.4: Example of a "discipline activation" influence: if a metallic material is chosen, the Metallic Structure Sizing tool is executed; if a composite material is chosen, the Composite Structure Sizing is executed. The activation condition is shown in the discipline block under the name of the discipline, after "Activation:".

4.1.2. Comparison of Dynamic MDAO Strategies

Architectural influences are introduced in the MDAO problem through "influence logic": the logic governing how the MDAO workflow modifies its behavior for each architecture instance it is evaluating. Influence logic covers any type of computational logic implemented for supporting architectural influences, such as if-statements to check whether a discipline should be activated, and the logic for determining the number of times a discipline should be executed.

Regardless of implementation or user interface, MDAO problems are formulated in three main steps [208] (see also Figure 1.16):

1. A tool repository is developed, with tools adhering to a CDS.
2. The MDAO problem is defined by selecting tools from the repository, establishing data connections, and selecting design variables, objectives and constraints.
3. The MDAO problem is made executable by applying an MDAO solution strategy: adding solvers and/or applying an MDAO architecture [210].

For SAO, the second step is extended to also include the implementation of the influence logic.

After formulation, the MDAO problem is deployed in some execution environment, such that it can be solved without user interaction in the architecture evaluator of the SAO loop (Figure 3.1). The formulation and deployment tasks are implemented by an MDAO formulator, which depending on where it is applied in the process, may require user interaction or not. Considering these aspects, the following high-level strategies for using MDAO for architecture evaluation are identified as part of this work:

1. *Single static* (Figure 4.5): a single static MDAO problem is defined. Influence logic is handled at the discipline-level (i.e. inside the disciplinary tools) instead of at the MDAO workflow level. The formulator is applied in the MDAO formulation phase (i.e. before running the SAO loop), and architecture evaluation consists of running

the static MDAO problem in the execution environment with the architecture instance as input. For this, no change in process and tooling is needed compared to non-SAO MDAO problems. However, this strategy is only possible if all architecture influences can be handled at the discipline-level, without requiring changes in tools and/or data connections for different architecture instances. This might require modification of disciplinary tools, which might be difficult or infeasible to manage in a cross-organizational team.

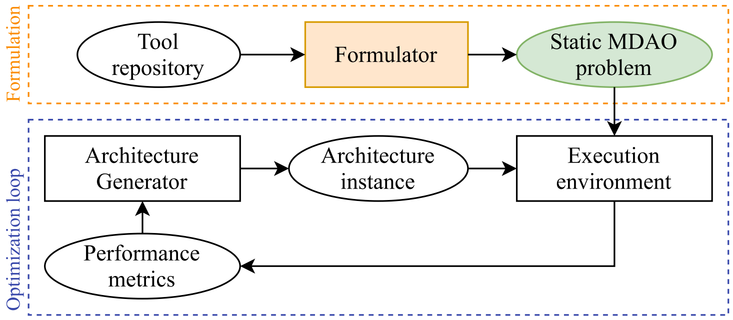


Figure 4.5: "Single static" dynamic MDAO strategy. The formulation phase (orange dashes) is executed before running the optimization loop (blue dashes). The optimization loop is executed without user interaction.

An example of an SAO problem solved with a single static MDAO workflow is the design of a business jet family [13, 16]: there, the architectural choices were regarding the sharing of aircraft components between family members, which were implemented using binary variables (0 = no sharing, 1 = sharing). The actual sharing logic was implemented in a dedicated tool in the workflow. This tool, however, always has the same inputs (sharing decision and potentially shared components) and outputs (overwritten component parameters), so a static MDAO problem was sufficient in that case. Another example is the comparison of various electrification levels of aircraft on-board systems [262]: here all the architectural changes are "contained" within one tool, allowing one static MDAO workflow being used for analyzing multiple architectures.

2. *Multi static* (Figure 4.6): multiple static MDAO problems are defined, each of which solves a predefined set of architecture instances. These sets are defined by enumerating all architectures and clustering by similar influence logic behavior, for example all architectures that lead to the same selection of tools and appearance of data connections. During execution of the SAO loop, the influence logic is then used by a router to determine which static MDAO problem should be used for the given architecture instance. This approach is limited by the number of architectures that can be considered as part of an architectural design space: if the number is too high, the overhead of implementing and managing the different workflows would become infeasible.

   A specific version of this strategy is one where the different architecture sets are defined and explored independently of each other, in which case the routing logic consists of the user triggering the independent execution. An example of this is the

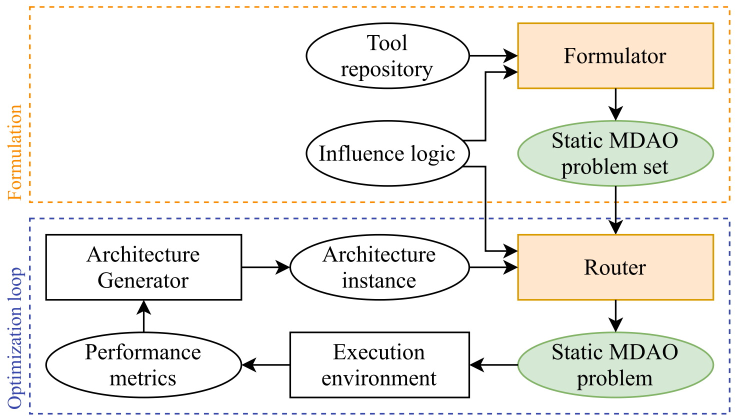


Figure 4.6: "Multi static" dynamic MDAO strategy.

strategy by FRANK ET AL. [130], who cluster architectures by active design variables and perform independent optimizations for each cluster, the results of which are later combined into a system-level Pareto front. BRUGGEMAN ET AL. [213] executed two independent static MDAO problems for two material selections (metal or composite), admittedly to compare their performances against a dynamic MDAO workflow. In this version of the strategy, all architectures are explored equally, which prevents the targeted exploration of more promising architectures and therefore potentially wastes computational resources.

3. *Single dynamic* (Figure 4.7): a single dynamic MDAO problem is defined. The dynamic MDAO problem implements influence logic directly in the workflow: influence logic is applied during execution, resulting in dynamically applied conditional data connections and tool execution. The dynamic MDAO workflow is setup in the formulation phase and executed with the architecture instance

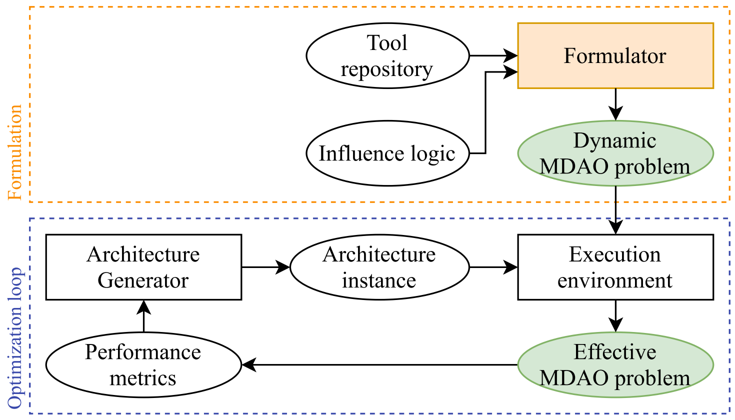


Figure 4.7: "Single dynamic" dynamic MDAO strategy.

as input during the SAO loop. The effectively established data connections and tool execution sequence is represented by the "effective" MDAO problem, a hypothetical problem representing the fact that the dynamic MDAO problem changes its behavior for different architecture instances. This strategy requires the MDAO formulation and execution process to also include influence logic in the MDAO problem.

This strategy was demonstrated by BRUGGEMAN ET AL. [213] with an MDAO problem that dynamically switches between two sub-workflows and adds/removes constraints based on an architectural decision (production method). The workflow was formulated before running the SAO loop, and the influence logic (choose the sub-workflow based on some architectural choice) was embedded in the executable workflow.

4. *On-demand* (Figure 4.8): automatically formulate a static MDAO problem for each architecture instance. This strategy requires a formulator that can perform the complete formulation and deployment process subject to influence logic, without user interaction.

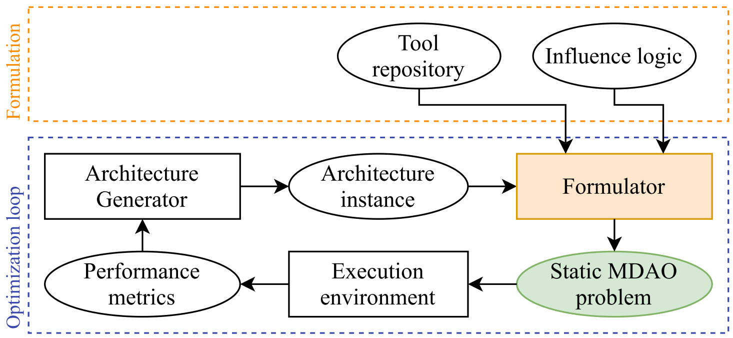


Figure 4.8: "On-demand" dynamic MDAO strategy.

Examples of this strategy are the jet engine architecture (Section 2.6) and hybrid-electric propulsion system (Section 3.5) application cases. For both, the static MDAO problems used for architecture evaluation were formulated and executed on-demand for each architecture instance. The static MDAO problems were implemented in OpenMDAO [248], and the OpenMDAO problems were defined from the architecture instances using extensive problem-specific Python code.

In another application, SONNEVELD ET AL. [214] dynamically changed a sub-workflow based on the number of ribs selected in an aileron, with the ribs indirectly influencing the number of material-thickness design variables through the torsion box geometry. No influence logic was explicitly embedded in the workflow: rather, the sub-workflow was generated on-demand by running a MDAO formulation tool that automatically formulates and executes the sub-workflow for the torsion box being designed.

Implementation for Collaborative MDAO For implementing the high-level strategies in collaborative MDAO, several aspects have to be considered: the use of a Central Data Schema (CDS), the black-box and distributed nature of the involved disciplinary tools, and the separation between the MDAO formulation and execution platforms.

The single static strategy requires a certain level of control over disciplinary tool development, something which might not be possible when using collaborative MDAO [206, 211], and might simply not be flexible enough for complex SAO problems. Therefore, single static is not deemed feasible for the general SAO case combined with collaborative MDAO.

The multi static strategy requires managing multiple workflows simultaneously, which will not be feasible above a relatively low number of workflows. As SAO problems may feature very large design spaces, this strategy is not deemed feasible for SAO either.

The on-demand dynamic strategy requires an automated formulator that also automatically deploys the workflow. As discussed in the introduction of this chapter, however, in collaborative MDAO the formulation phase is usually separate from the execution phase, and some manual steps may be needed before the workflow as deployed in the Process Integration and Design Optimization (PIDO) environment can be executed. These manual steps are what currently prevents the application of the on-demand strategy for collaborative MDAO.

Therefore, for the implementation in this research, the *single dynamic* workflow strategy (Figure 4.7) is chosen. The following section presents the implementation of this strategy in a collaborative MDAO tool.

4.1.3. Implementation in MDAx

The MDAO Workflow Design Accelerator (MDAx) is a workflow modeling tool developed by the DLR [19, 28]. It features an interactive GUI for creating and manipulating MDAO workflows in an XDSM [258] representation, automatically connecting declared tool input and output. For this, all tool inputs and outputs are declared in terms of a Central Data Schema (CDS). It builds on the graph-based MDAO definition principles developed by van Gent and implemented in the open-source KADMOS tool [209]. Both MDAx and KADMOS implement the MDAO workflow formulation phase as discussed in the previous section, and delegate execution to Process Integration and Design Optimization (PIDO) platforms. MDAO problem visualization and limited interactivity for KADMOS was provided by VISTOMS [263]; MDAx provides its own interactive GUI for creating and inspecting MDAO workflows. Further differences between the two tools are discussed in [19].

To deploy and execute the workflow, MDAx supports two export formats: the CMDOWS format [264] and a Remote Component Environment (RCE) [265] workflow. RCE is an open-source PIDO tool also developed by the DLR that enables distributed and cross-organizational execution of disciplinary tools in simulation workflows. In this work, the single dynamic MDAO strategy is applied and implemented in MDAx (excluding the GUI) and MDAx' RCE export (the CMDOWS format and MDAx' CMDOWS export is not modified).

In the RCE export of an MDAx workflow, the disciplinary analysis tool is wrapped by various scripts that implement various dynamic MDAO mechanisms. Figure 4.9 shows

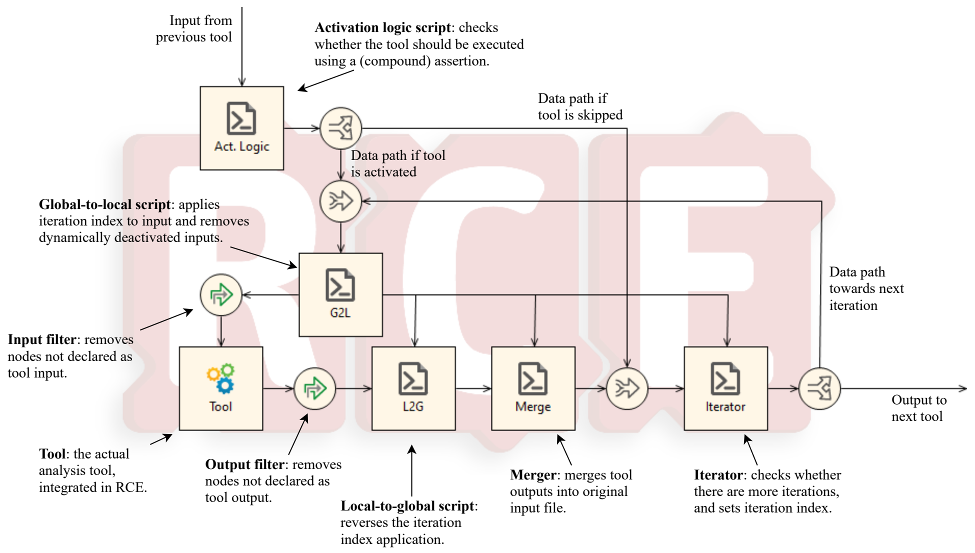

Input from previous tool
Activation logic script: checks whether the tool should be executed using a (compound) assertion.
Act. Logic
Data path if tool is skipped
Data path if tool is activated
Global-to-local script: applies iteration index to input and removes dynamically deactivated inputs.
G2L
Data path towards next iteration
Input filter: removes nodes not declared as tool input.
Tool
L2G
Merge
Iterator
Output to next tool
Tool: the actual analysis tool, integrated in RCE.
Output filter: removes nodes not declared as tool output.
Local-to-global script: reverses the iteration index application.
Merger: merges tool outputs into original input file.
Iterator: checks whether there are more iterations, and sets iteration index.

Figure 4.9: Annotated detail of MDAx RCE export showing all possible wrapper mechanisms around a tool: activation logic script, G2L and L2G scripts, input and output filters, merger and iterator.

a detailed view of such an export, with the wrapped analysis tool on the lower left side of the figure. The input and output filter blocks are retained from the non-dynamic MDAx implementation, preparing the tool input and output such that only declared inputs reach the tool and only declared outputs are merged back into the CDS file (in the merger script) [19]. A summary of the mechanisms for supporting the dynamic MDAO architecture influences follows. For more details refer to [261].

Discipline Activation Disciplines can be assigned an *activation assertion*, which when evaluated to `true` activates the discipline, and when `false` deactivates the discipline (a deactivated discipline is skipped during execution). The assertion is executed right before executing the discipline, so disciplines may be activated and/or deactivated various times during the execution of the dynamic MDAO workflow.

Various assertion primitives are implemented, and assertions can be recursively combined into compound assertions using `and` and `or` operators. Assertion primitives include checking for CDS variable existence, whether variables have a value, or checking if the value of a variable adheres to some condition (including numerical lower than, equal to, and greater than comparisons). Also checking the number of nodes that exists at some xpath[1] (currently MDAx only supports XML as data storage format) is possible.

In the exported RCE workflow (see Figure 4.9), an activation logic script is added to every disciplinary tool that contains an activation assertion. In addition to the current CDS file, this script outputs whether or not the assertion was satisfied, which is then used in a "switch" component to execute or bypass the tool execution block based on the activation assertion.

[1] https://www.w3.org/TR/xpath/

Discipline Repetition Disciplines can be assigned a *repetition* property, which determines the number of times a tool should be executed based on the CDS. The number can either come from the number of nodes referred to by an xpath expression, or the numerical value of a variable in the CDS.

In the RCE workflow, a Global-to-Local (G2L) conversion step converts from the iteration-index-aware workflow-level ("global") input to index-agnostic tool-level ("local") input (see Figure 4.9). This ensures that at the tool-level, each repetition receives the same input variables. For example, a repeated engine sizing tool might need the geometry of each respective engine, defining its input to be located at `/inputs/engine/geometry` in the CDS. However, the CDS contains the different engine geometries at `/inputs/engine_i/geometry` (where `i` is the index of the engine being processed in the current iteration): the G2L conversion translates this "global" CDS to the "local" CDS as needed by the tool. The result is that the input/output definition of the repeated tool can stay static (and therefore black-box), even though it is applied to different instances of input/output data.

Data Connection and Conditional Variables Because MDAx is based on automatically deriving data connections from tool input/output definitions, *data connection* changes are implemented by dynamic modification of these definitions. Conditional input/output variables are supported using the same mechanism. Dynamically modifying tool input/output is done by associating assertions to xpaths that refer to (groups of) variables in the tool's input or output definition. These assertions are the same as implemented for supporting discipline activation. The mechanism has only been implemented for dynamic modification of tool inputs, however can also be implemented for tool outputs in the future.

In the RCE export (see Figure 4.9), dynamic modification of tool inputs is implemented in the G2L converter block, which simply removes variables in the input file if the associated assertion resolves to `true`. Dynamic modification of tool outputs would be implemented in the local-to-global block.

This section has presented how dynamic MDAO principles can be implemented in an MDAO workflow, such that the workflow can be used as the architecture evaluator in an SAO loop. The following section presents how a two-way data connection can be established between the architecture generator and the dynamic collaborative MDAO workflow.

4.2. ARCHITECTURE DATA PROPAGATION

Previous work on connecting system architecture models to MDAO workflows has only supported linking numerical parameter values, for example [77, 200, 203, 205]: the system architecture is kept static, and if the system architecture would have changed, the MDAO workflow would have had to be updated manually. This section extends previous work by enabling the communication of any relevant information about system architecture (such as the selection and instantiation of components, function allocations, and port connections), so that it can be used by the MDAO workflow to dynamically modify its behavior (see Section 4.1) based on the system architecture being evaluated.

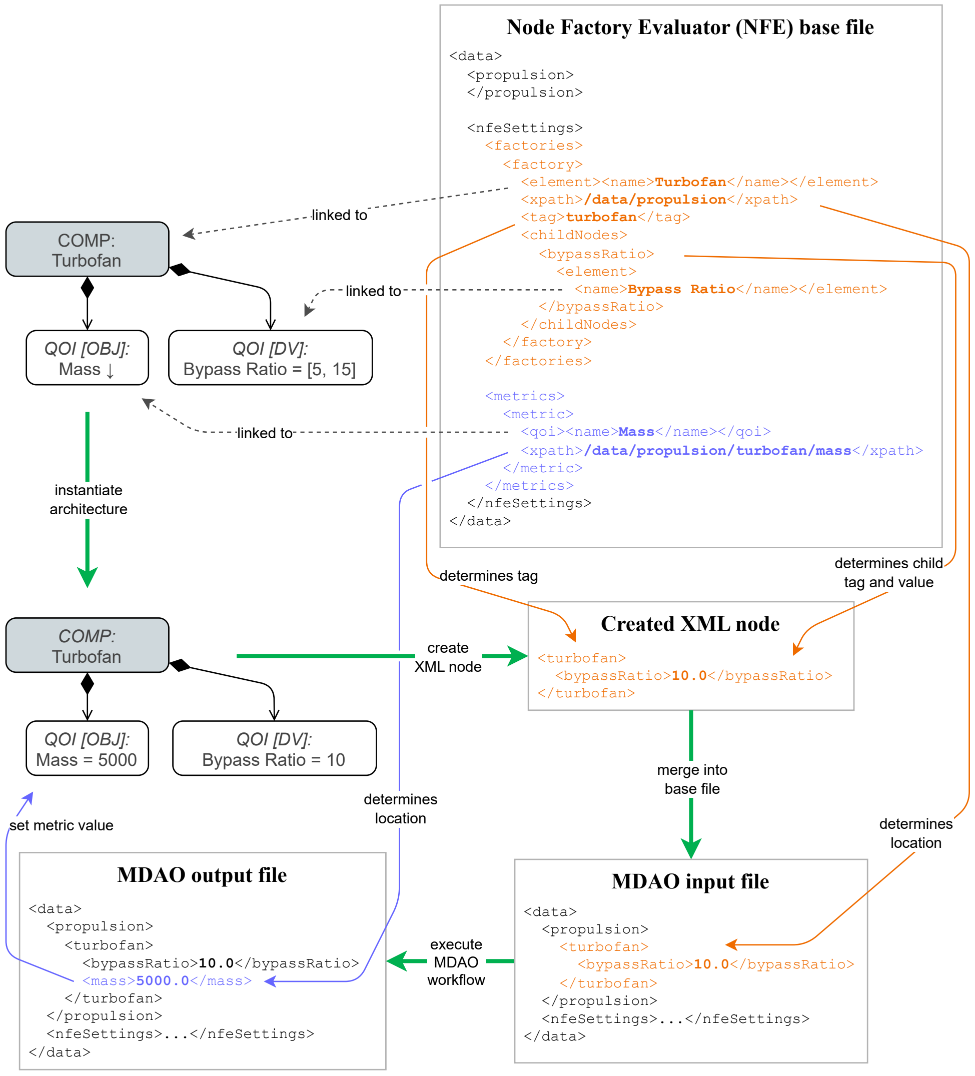
Architecture data propagation from architecture instances generated by ADORE to the Central Data Schema (CDS) of a collaborative MDAO workflow is implemented by the *Node Factory Evaluator* (NFE). By basing its implementation on the Class Factory Evaluator (CFE; see Section 3.3.2), it supports transmitting all available information. Additionally, the translation rules are defined in the base file itself, thereby effectively making the user-interface that input file: this requires less context-switching (e.g. between the workflow and some GUI) when defining and testing node factories.

The NFE works by defining node and metric factories. Node factories determine how architecture elements (e.g. components, QOIs) are transformed into XML nodes, which are then merged into the base file to obtain the input MDAO file. Metric factories define where values for output metrics can be read from the CDS file. If a value is not found, the metric is assigned NaN (not-a-number). In addition to the factories themselves, the NFE also needs as input the xpath where the factories are defined, as this might be specific to the used CDS.

Figure 4.10 shows the working principle of the NFE by an example. It shows how the NFE base file contains the node and metric factory definitions, with the xpath to the factories being `/data/nfeSettings`. A node factory consists of:

1. an *element* definition, linking the factory to an architecture element;
2. the *xpath* where the resulting XML node is merged into the base file;
3. the *tag* of the created node; and
4. *value, child node, or attribute* definitions, determining the contents of the node.

A metric factory only consists of an element definition and an xpath. This flexibility of defining the xpath, tag, attributes, and contents (node value or child nodes) for each factory and to define where factories are defined allows close adherence to the CDS format. The node factory is used to transform the Turbofan component of the created architecture instance to a `turbofan` node with a `bypassRatio` child node, of which the value comes from the Bypass Ratio design variable. The created node is then merged into the base file at the specified xpath, resulting in the MDAO input file. The metric factory is applied to the MDAO output file: the Mass QOI of the Turbofan is assigned the value at the specified location.



Node Factory Evaluator (NFE) base file

```
<data>
  <propulsion>
  </propulsion>

  <nfeSettings>
    <factories>
      <factory>
        <element><name>Turbofan</name></element>
        <xpath>/data/propulsion</xpath>
        <tag>turbofan</tag>
        <childNodes>
          <bypassRatio>
            <element>
              <name>Bypass Ratio</name></element>
          </bypassRatio>
        </childNodes>
      </factory>
    </factories>

    <metrics>
      <metric>
        <qoi><name>Mass</name></qoi>
        <xpath>/data/propulsion/turbofan/mass</xpath>
      </metric>
    </metrics>
  </nfeSettings>
</data>
```

Created XML node

```
<turbofan>
  <bypassRatio>10.0</bypassRatio>
</turbofan>
```

MDAO input file

```
<data>
  <propulsion>
    <turbofan>
      <bypassRatio>10.0</bypassRatio>
    </turbofan>
  </propulsion>
  <nfeSettings>...</nfeSettings>
</data>
```

MDAO output file

```
<data>
  <propulsion>
    <turbofan>
      <bypassRatio>10.0</bypassRatio>
      <mass>5000.0</mass>
    </turbofan>
  </propulsion>
  <nfeSettings>...</nfeSettings>
</data>
```

Figure 4.10: Example of the Node Factory Evaluator (NFE) applied to the evaluation of the Turbofan component with two Quantities of Interest (QOIs): a bypass ratio design variable and a mass minimization objective. The node and metric factories are linked to elements in the ADORE architecture model. After an architecture has been instantiated, the node factory is applied, resulting in the MDAO input file. The output metric value (Mass) is read from the MDAO output file.

4.3. PROCESS INTEGRATION

The previous sections have presented how to define dynamic MDAO workflows and execute them in Process Integration and Design Optimization (PIDO) environments, and how to propagate data from the architecture generator (ADORE) to the MDAO workflow. This section discusses how ADORE can be integrated in a PIDO environment, so that the complete SAO loop can be executed from that same environment. Additionally, it presents how the SAO problem can be defined without requiring programming or local installation of ADORE. The principles of integrating SAO in PIDO environments are presented in Section 4.3.1. Section 4.3.2 then presents the implementation into ADORE and a PIDO environment.

4.3.1. THE SAO ASK-TELL PATTERN

The architecture generator consists of the optimization algorithm and the architecture generator step. For collaborative MDAO, this is extended by either direct translation of the architecture instance to XML (or whatever format the MDAO workflow uses) or translation using the Node Factory Evaluator (NFE; see Section 4.2). To integrate these three steps into the PIDO environment, where the PIDO environment has control over what is executed when, the ask-tell pattern is applied.

The ask-tell pattern comes from numerical optimization and decouples the optimization algorithm from the evaluation function [227]: an external loop "asks" the optimizer for a new design vector to be evaluated, and after evaluation "tells" the optimizer the results. This approach has three main advantages compared to letting the optimizer drive the optimization loop:

- The execution of the optimizer and the evaluation code are decoupled. This makes it easier to implement advanced optimization strategies like multi-start or replacing the optimization algorithm during the optimization (for example to start with a global optimization algorithm to approximate the optimum, and finish with a local optimization algorithm to increase accuracy).
- The decoupling also enables executing the optimizer and evaluation code on different machines, which can be used to offer the optimizer as a service and/or for protecting intellectual property rights.
- Optimizer state must be stored in a well-defined data structure. This enables storage of intermediate results and restart, which is useful when running expensive evaluations.

For integration of SAO and collaborative MDAO, the principle of decoupling is applied to the interface between the MDAO workflow and the architecture generator (including the NFE). The resulting so-called SAO ask-tell pattern is listed in Algorithm 1. The STATE variable contains the union of all inputs to the ask and tell functions (except n_{batch} and OUTPUTFILES), consisting of the architecture design space model and the state and configuration of the optimization algorithm and Node Factory Evaluator (NFE). STATE should be stored in a file to ensure that the steps can be executed decoupled and without requiring the architecture generator to run in the background.

Algorithm 1 The collaborative SAO optimization loop. The "ask" and "tell" functions are detailed in Algorithms 2 and 3, respectively. The STATE variable contains the union of all inputs to the ask and tell functions, except n_{batch} and OUTPUTFILES.

```
Require: $n_{\text{batch}}$, state
Ensure: state, archList
 1: repeat
 2:     inputFiles, state ← ask (state, $n_{\text{batch}}$)          ▷ Ask architecture generator
 3:     outputFiles ← evaluate (inputFiles)           ▷ Execute collaborative MDAO problem
 4:     state ← tell (outputFiles, state)              ▷ Tell results to architecture generator
 5: until len (inputFiles) = 0
 6: archList ← getArchList (state)         ▷ Extract generated and evaluated architectures
```

Algorithms 2 and 3 present the ask and tell steps in more details, respectively. Compared to the original ask-tell pattern, a buffer is added between the optimization algorithm and the architecture generation step, to support evaluating a different number of architectures than suggested by the optimization algorithm. For example, the first generation of an evolutionary algorithm will suggest to evaluate n_{doe} points, whereas collaborative MDAO workflows can typically only evaluate one input file at a time.

Algorithm 2 "Ask" step of the collaborative SAO optimization loop, using the NFE for providing input data for collaborative MDAO. Requesting multiple input files for parallel evaluation is supported ($n_{\text{batch}} > 1$). An empty list of input files means that the optimization has finished.

```
Require: $n_{\text{batch}}$, buffer, optimizer, optState, designSpaceModel, archList, staticInput,
    nodeFactories
Ensure: inputFiles, buffer, optState, archList
 1: if len (buffer) = 0 then                                        ▷ If buffer is empty
 2:     buffer, optState ← askOpt (optimizer, optState)  ▷ Ask the optimization algorithm
 3: end if
 4: $x$ ← get (buffer, $n_{\text{batch}}$)            ▷ Get next $n_{\text{batch}}$ design points from the buffer
 5: inputFiles ← []                                            ▷ Clear list of input files
 6: for $x_i$ in $x$ do
 7:     arch ← generate (designSpaceModel, $x_i$)            ▷ Generate architecture instance
 8:     archList ← {archList, arch}                          ▷ Store generated architecture
 9:     inputFile ← apply (arch, staticInput, nodeFactories)             ▷ Generate input file
10:     inputFiles ← {inputFiles, inputFile}                     ▷ Add to list of input files
11: end for
```

Algorithm 3 "Tell" step of the collaborative SAO optimization loop, using the NFE for reading output data from collaborative MDAO. Multiple output files can be supplied, the amount set by n_{batch}.

```
Require: outputFiles, $n_{\text{batch}}$, buffer, optimizer, optState, designSpaceModel, archList,
    staticInput, metricFactories
Ensure: buffer, optState
 1: results ← []
 2: for outputFile in outputFiles do                                ▷ Loop over output files
 3:     arch ← read (outputFile, metricFactories, archList)
 4:                     ▷ Match output file to an architecture and interpret output metrics
 5:     $x, \delta, f, g \leftarrow$ extract (arch)           ▷ Extract evaluation results from the architecture
 6:     results ← {results, $[x, \delta, f, g]$}                    ▷ Append to list of results
 7: end for
 8: buffer ← apply (buffer, results)                 ▷ Update buffer with evaluation results
 9: $x_{\text{next}}$ ← get (buffer, $n_{\text{batch}}$)               ▷ Get new design points from the buffer
10: if len ($x_{\text{next}}$) = 0 then           ▷ If the buffer contains new design points anymore
11:     optState ← tellOpt (buffer, optimizer, optState)              ▷ Tell results to optimizer
12:     buffer ← []                                                         ▷ Clear the buffer
13: end if
```

4.3.2. IMPLEMENTATION OF THE SAO ASK-TELL PATTERN

This section presents the implementation of the SAO ask-tell pattern in ADORE and RCE, DLR's open-source PIDO environment [265]. The implementation into RCE can be applied to other PIDO environments using the same mechanisms.

Implementation in ADORE ADORE uses SBArchOpt (see Section 3.3.3) for optimization, which is based on pymoo [244]. Pymoo optimization algorithms are structured such that they support the ask-tell pattern[2]. Optimization state is managed by ADORE and stored in an ADOREOPT file, which stores the following inputs to the ask and tell steps (Algorithms 2 and 3, respectively):

- The ADORE model, including generated ADORE architectures.
- Selection, configuration, and state of the optimization algorithm.
- Selection and configuration of the architecture translator.
- The design points buffer, including evaluation status and iteration counter.

Storing all above in a file also allows transitioning from the GUI where the SAO problem is configured, to the PIDO environment where the SAO problem is executed.

[2]https://pymoo.org/algorithms/usage.html#nb-algorithms-ask-and-tell

In the GUI, the process to define the SAO problem is then as follows (see also Section 3.3.1):

1. Model the architecture design space (see Section 3.4).
2. Encode the design space as an optimization problem, optionally freezing design variables and/or reassigning output metric roles.
3. Select and configure the optimization algorithm, one of (see Chapter 2): Design of Experiments, NSGA-II, or Bayesian Optimization algorithm.
4. Select and configure the evaluation interface, one of (see Section 3.3.2): XML NFE (see Section 4.2), XML or JSON file-based serialization.
5. Export the ADOREOPT file.

Note that the available optimization algorithms and evaluation interfaces can be extended, as long as they are compatible with the SAO ask-tell pattern.

Implementation in PIDO Environments After the ADOREOPT file has been created, the process moves to the PIDO environment. In this discussion that is RCE, however the principles also apply to others. In the PIDO environment, a new computational block is integrated that implements the architecture generator step. This architecture generator block is connected to the rest of the MDAO workflow: the architecture generator block provides as *output* the generated input file to the blocks comprising the MDAO workflow, and receives as *input* the file resulting from the MDAO workflow blocks. Combining these two connections actually means that the architecture generator block implements a *tell-ask* step, that is: first, outputs are told to (Algorithm 3) and then, new inputs are asked from (Algorithm 2) the architecture generator. In the current implementation, the collaborative MDAO workflow can only process one input file at a time, so $n_{\text{batch}} = 1$ in the ask step.

Figure 4.11 shows all the steps involved in the collaborative SAO loop, including the NFE (Section 4.2) and a dynamic MDAO workflow formulated in MDAx (Section 4.1). After execution, the ADOREOPT file can be opened in the ADORE GUI to inspect or export the generated and evaluated architectures.

When first starting the workflow, there are no outputs so there is some starting file that will be passed to the tell-ask block to start the collaborative SAO loop. If the NFE is used, the start file is actually the base file (see Figure 4.10) which will subsequently be stored in the ADOREOPT file. This principle of providing the base file when executing the workflow (rather than when defining the SAO problem in the ADORE GUI) enables quick development and testing of the defined node and metric factories, since the results are observed in the same environment as where the base file is defined.

The tell-ask block runs some Python function in ADORE, and therefore requires that ADORE is installed in the Python environment. In some project contexts it might not be desirable or possible to share the ADORE code to the same machine that is running the SAO problem. This scenario is supported by the fact that the input to and output from the tell-ask block only consist of files to be transferred: the tell-ask block can thus be hosted on another machine than the one running the workflow according to collaborative MDAO principles.

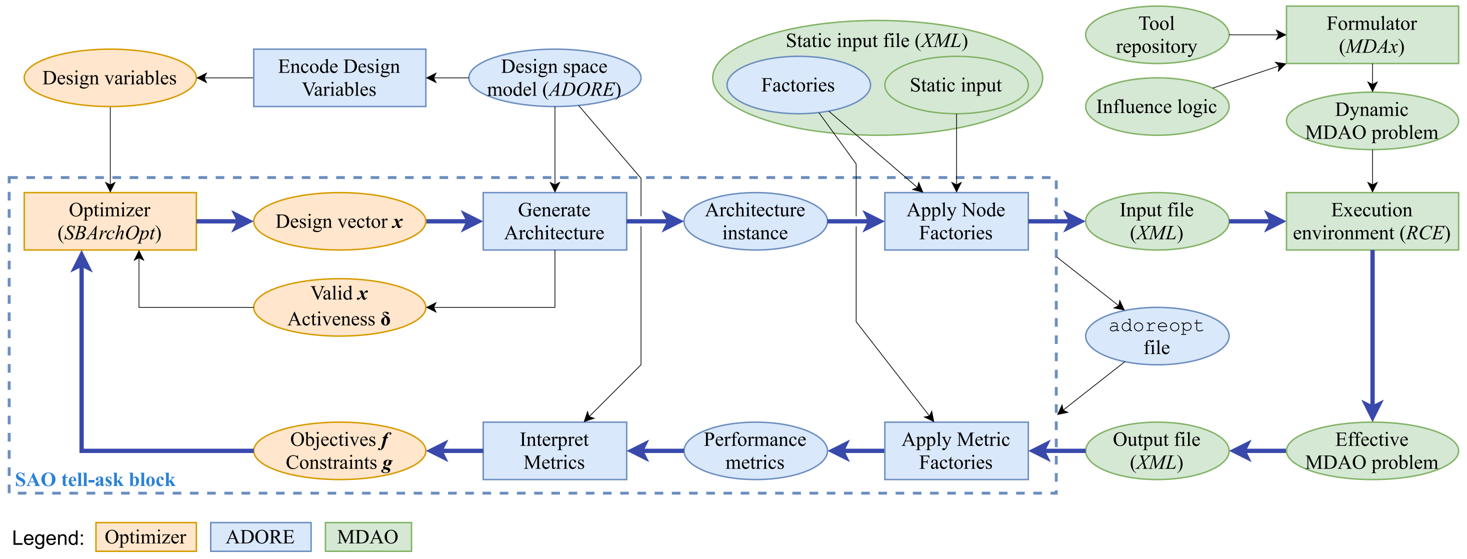


Figure 4.11: Collaborative SAO loop, including the NFE and a dynamic MDAO workflow formulated in MDAx. Tools or technologies implementing the specific steps are shown in italics between brackets. The optimization loop is highlighted by bold blue arrows.

4.4. APPLICATION CASE III: MULTI-STAGE LAUNCH VEHICLE

This application case presents the design of a multi-stage launch vehicle architecture using dynamic MDAO, the Node Factory Evaluator (NFE), and the integration of ADORE in RCE as presented in the preceding sections. The ADORE model and resulting optimization problem is presented in Section 3.6.1. The SAO problem involves the choice of number of stages, stage-level choices (number of engines, engine types, and stage length), and rocket geometry choices (head shape and length-to-diameter ratio). It is a multi-objective problem, with the goal to maximize payload mass and minimize cost for a given target orbit altitude, subject to structural and payload volume constraints.

Dynamic MDAO Workflow The MDAO workflow for sizing and evaluating the generated rocket architectures contains the following disciplinary tools, shown also in Figure 4.12:

- *Propulsion* calculates the thrust produced by the selected engines for a given stage, and is implemented by either:
  - *Solid propulsion* for either the SRB, P80 or GEM60 engines; or
  - *Liquid propulsion* for either the VULCAIN, PS68 or S_IVB engines; also calculates nozzle expansion ratio.
- *Geometry* calculates some geometrical parameters like fuel and oxidizer (if liquid propulsion is used) tank surface or volume, and the rocket head surface and volume. For this it needs inputs such as the engine types, stage lengths, length-to-diameter ratio, and head shape and associated parameters.
- *Propellant mass* calculates how much propellant each stage can store, given engine types and tank volumes.
- *Structural mass* calculates masses of various structural components for each stage, such as outer structure, tank structure, and masses of other systems.
- *Trajectory* simulates the trajectory taken by the rocket given the masses, geometry (to calculate drag), thrust, and engine types, and solves for the maximum payload mass that can be taken while still reaching the requested orbit altitude. If the orbit cannot be reached even without payload, the payload mass is set to zero.
- *Cost* calculates the cost to manufacture and launch the rocket, based on statistical correlations on the engine types and propellant and structural masses.
- *Structural constraint* calculates whether the structure would fail during launch, by comparing the maximum dynamic pressure achieved during the solved trajectory to the maximum permissible dynamic pressure.
- *Payload volume constraint* calculates whether enough volume is available in the rocket head by comparing the head volume to the maximum payload mass multiplied by an assumed payload density.

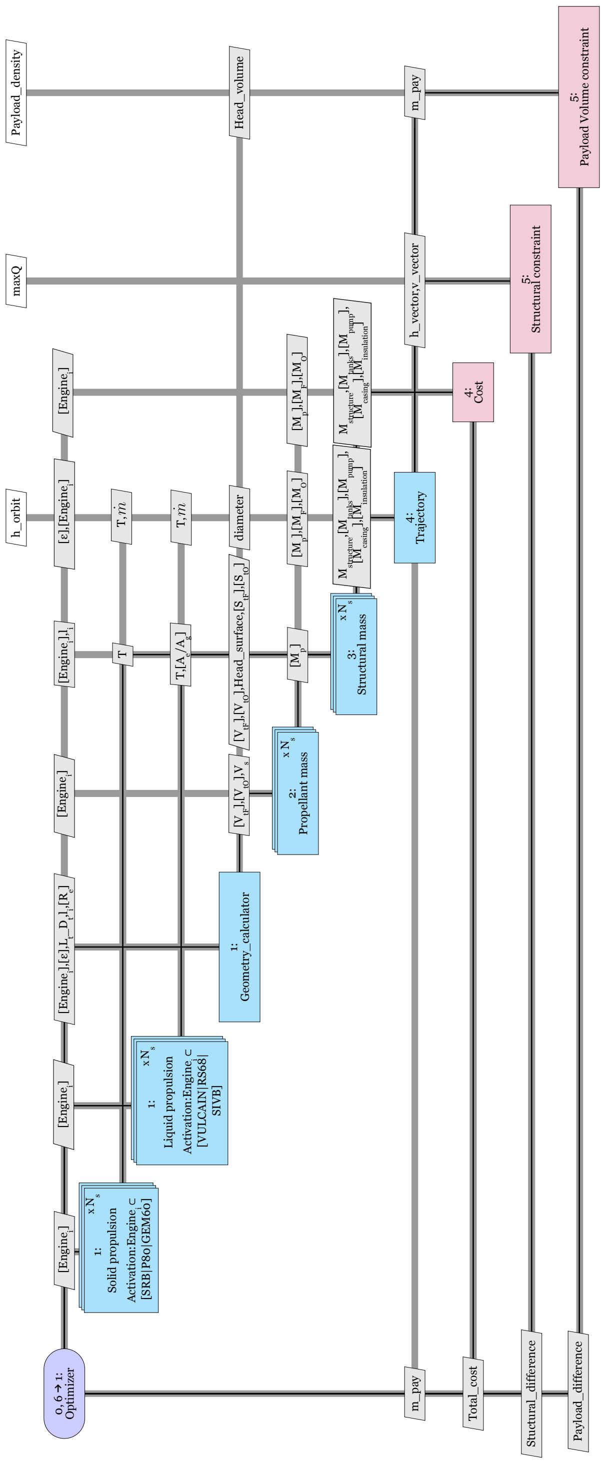


Figure 4.12: XDSM view of the multi-stage launch vehicle problem, showing discipline repetition due to number of stages (N_s in upper right corner), discipline activation, conditional variables (variables wrapped in square brackets), and data connection influences (bold/italic variables).

The propulsion, propellant mass, and structural mass disciplines all act on one stage at a time and therefore need to be repeated according to the number of selected stages (discipline repetition influence). Their inputs and outputs are also only applied to the location within the used Central Data Schema (CDS) that applies to the respective stage. The propulsion discipline is implemented by a solid or liquid propulsion tool depending on the type of engine selected for each stage (discipline activation influence). Various tool inputs and outputs exist conditionally depending on architectural choices, such as propellant masses, tank geometry, and nozzle expansion ratio. An example of the CDS used in the workflow is listed in Appendix F.

Figure 4.12 shows the problem in XDSM [258] view, including architectural influences as presented in Section 4.1.1. The implementation of the MDAO disciplines and RCE workflow exported from the MDAx model of the dynamic MDAO workflow is available open-source[3].

Connection to ADORE The connection between the ADORE model and the dynamic MDAO workflow is established using the NFE. Appendix F shows the start file including NFE node and metric factories. The start file does not contain any nodes of the CDS, because the complete data structure is constructed only by the node factories. Each factory adds a node with some contents at some xpath. Since factories are executed in the order they are defined, factories can add nodes to nodes previously constructed by other factories. This mechanism is used to add geometry and engine selection information to the different "Stage" nodes.

The MDAO workflow is implemented in an RCE workflow, which also contains the ADORE ask-tell block introduced in Section 4.3.

Optimization Results The optimization problem is solved using NSGA-II (evaluation is not expensive) with a population size of 150 and 30 generations (a budget of 4500 function evaluations). Figure 4.13 shows the feasible results of this optimization, which consists of 763 design points, of which 32 are in the Pareto front. Figure 4.14 presents various categorizations of the Pareto front. It shows a clear separation by number of stages, with more stages used for larger rockets (rockets that can carry more payload and are more expensive). Larger rockets also use solid (higher power output) over liquid propulsion (lighter), and tend to use an elliptical head shape (lower drag) versus semisphere (more payload volume) for smaller rockets.

[3]https://github.com/raul7gs/Space_launcher_benchmark_problem

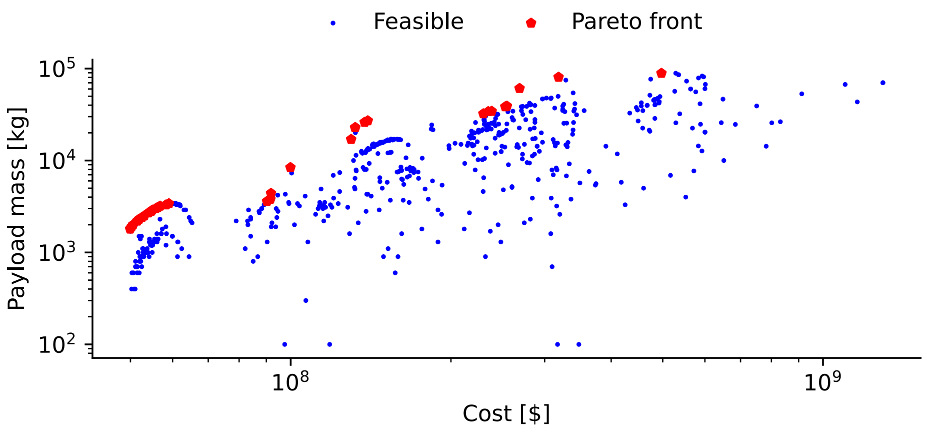
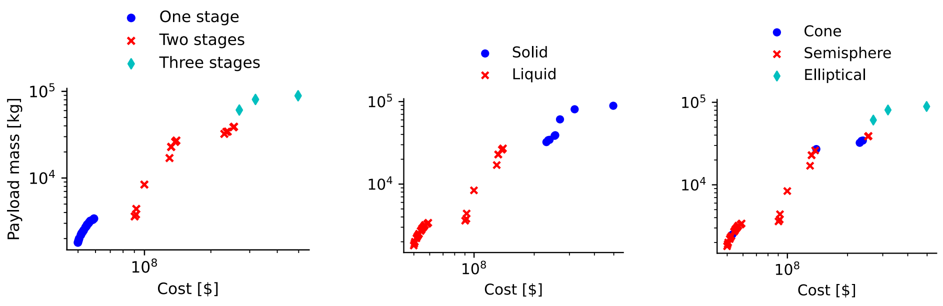


Figure 4.13: Results of the multi-stage launch vehicle SAO problem, implemented using NFE and dynamic MDAO. Payload mass is to maximized; cost to be minimized.



(a) By number of stages. (b) By 1st stage propulsion. (c) By rocket head shape.

Figure 4.14: Various categorizations of the Pareto front (maximizing payload mass vs minimizing cost) of the multi-stage launch vehicle SAO problem.

4.5. CHAPTER CONCLUSIONS

In this chapter, the connection of System Architecture Optimization (SAO) to collaborative Multidisciplinary Design Analysis and Optimization (MDAO) as developed in this research has been presented.

- Four high-level strategies have been identified for implementing MDAO for SAO (Section 4.1): single static, multi static, single dynamic, and on-demand. This work implements the single dynamic strategy: a single MDAO workflow dynamically modifying its behavior for each architecture instance being evaluated. The following architecture influences are supported:
  - Discipline activation: the dynamic activation or skipping of disciplinary tools as determined by some assertion acting on the Central Data Schema (CDS).
  - Discipline repetition: repeating a tool according to the number of times some node in the CDS is repeated.
  - Data connection and conditional variables: dynamically rerouting data connections or variables that may or may not exist due to some architecture decision.

  Previous work has either not supported dynamic behavior at all, or has only supported it to a limited extent.
- Implementing a way to transfer all relevant information from an architecture instance (such as the selection and instantiation of components, function allocations, and port connections) as generated by ADORE to the CDS as used in the collaborative MDAO workflow (Section 4.2).

  This is done using the Node Factory Evaluator (NFE), which uses node factories to dynamically add nodes to the CDS file based on architectural elements, and metric factories to read resulting performance metrics from the CDS file. Previous work has only supported the transfer of numerical parameter values, assuming a static system architecture and CDS structure.
- Decoupling the architecture generator from the evaluation using an SAO tell-ask pattern (Section 4.3). The Process Integration and Design Optimization (PIDO) environment coordinates when the architecture generator is "asked" for a new architecture to evaluate (converted to the CDS using the NFE), and when the architecture generator is "told" the results of the MDAO workflow. This enables the integration of the architecture generation step into the PIDO environment where the MDAO workflow is deployed.

  Additionally, allowing the user to define the SAO problem and configure the optimization algorithm from the GUI enables smooth switching between ADORE and the PIDO environment.

These three developments enable collaborative MDAO to be used for evaluating architectures in an SAO loop: the workflow is available for all architecture instances, sensitive to architecture design choices, and can be executed automatically for every generated architecture.

The collaborative SAO approach is demonstrated using a multi-stage launch vehicle problem (Section 4.4). The application case shows the development of a dynamic MDAO workflow contains all identified architecture influences, and a demonstration of the NFE for converting the architecture instances generated by ADORE to the CDS. The problem was solved using NSGA-II, resulting in a Pareto front of optimal architectures trading-off cost and maximum payload mass.

The main difference of the dynamic collaborative MDAO approach (Section 4.4) compared to the locally-integrated Python-only implementation (Section 3.6.1) is that it allows for the distributed management, development, and execution of disciplinary analysis tools [36]. For projects involving different organizations, this might be necessary due to intellectual property management constraints (preventing centralized execution of tools) and/or due to resource constraints (preventing the problem-specific modification of tools). However, if such constraints do not apply to an SAO problem, and if it would be feasible to manage, develop, and execute all analysis tools by one person, on one machine, then a locally-integrated approach might be more suitable. A locally-integrated MDAO workflow requires less tool and interface (such as the CDS) management overhead, because tools and interfaces are managed by the workflow integrator only.

With these results, the third sub-objective has been achieved:

3. Develop a methodology for leveraging collaborative MDAO for quantitative architecture evaluation, by:
   (a) enabling changing MDAO workflow behavior due to architectural changes;
   (b) propagating architecture data to the collaborative MDAO workflow; and
   (c) integrating SAO within a collaborative MDAO process integration platform.

5 Conclusions & Recommendations

This dissertation has introduced a methodology and tools for defining and solving System Architecture Optimization (SAO) problems. SAO automates the system architecting phase of a Model-Based Systems Engineering (MBSE) process, by modeling the architecture design space and then transforming this model into a numerical optimization problem that can be solved by appropriate optimization algorithms. The architecture design space should be defined based on system functions to promote unbiased selection of form and enable traceability from system requirements, and it should be possible to define architecture-specific design parameters in addition to architectural choices. Applying numerical optimization algorithms prevents the need to fully enumerate the potentially large combinatorial design space. The optimization algorithms, however, should be able to deal with the mixed-discrete, hierarchical, multi-objective, expensive, black-box, constrained (due to design constraints and hidden constraints) nature of SAO problems.

Design vectors generated by the optimization algorithm in the SAO design loop are transformed to architecture instances, which are automatically evaluated by multidisciplinary system-level codes. This automatic, quantitative, multidisciplinary evaluation of system performance further reduces design bias, enables searching a larger design space, and ensures a realistic, balanced design is the result. Architecture evaluation should be flexible enough to ensure relevant performance metrics are available for all generated architecture instances, and it should be sensitive to relevant architectural choices and design parameters. Collaborative Multidisciplinary Design Analysis and Optimization (MDAO) enables computational coupling of heterogeneous, cross-organizational engineering disciplines, thereby supporting the multidisciplinary nature of architecture evaluation. Combining these observations, the objective of this research was the following:

> to enable practical usage of System Architecture Optimization (SAO), by developing a methodology for formulating SAO problems, providing algo-

rithms for solving these problems, and by using collaborative MDAO to evaluate architecture instances.

This chapter reviews the achievements of this objective in Section 5.1, recommends topics to move research into and application of SAO forwards in Section 5.2, and provides a vision for the future of SAO in Section 5.3.

5.1. CONCLUSIONS

To answer the main research question:

How can System Architecture Optimization (SAO) improve the design space exploration and optimization of complex systems? By modeling and formally defining SAO problems using the Architecture Design Space Graph (ADSG) as implemented in ADORE, applying Bayesian Optimization (BO) adapted for SAO to solve the problems, and supporting the definition of dynamic collaborative MDAO workflows connected to the architecture generator to evaluate architecture instances.

More details are provided by the answers to the three sub-questions below.

How to formulate SAO problems based on system functions? By using a layered approach consisting of:

- the Design Space Graph (DSG) (available open-source as ADSG CORE[1], resulting from this work), which models hierarchical selection and connection choices and provides algorithms for encoding these choices into design variables, and metrics into objectives and constraints for formal definition of the optimization problem;
- the Architecture Design Space Graph (ADSG), which extends the DSG with node types and rules specific to system architecting, such as functions, components, and ports; and
- ADORE (Architecture Design and Optimization Reasoning Environment), which provides a user-friendly graphical user interface way to define the ADSG, and various interfaces for connecting to architecture evaluation code and optimization algorithms, enabling integration into the MBSE process.

A bottom-up process for modeling SAO problems using ADORE and the ADSG was defined, including guidelines for formulating functions, fulfilling functions using components, decomposing functions, using subsystems and architectural constraints, and for defining performance metrics. It was demonstrated that the bottom-up process (which starts from boundary functions and mainly uses function-component zigzagging) results in a more natural approach compared to top-down processes (which mainly use function decomposition), such as enabled by tree-based models.

A hybrid-electric propulsion architecture optimization problem demonstrated the developed approach, featuring an ADORE model with function allocation choices,

[1]https://adsg-core.readthedocs.io/

choice hierarchy, a subsystem, choice constraints, and Quantity of Interest (QOI) definitions. Architectures were evaluated using a locally-executed and on-demand constructed MDAO problem, which was connected to ADORE using the Class Factory Evaluator (CFE) interface. A second demonstration showed using three SAO problems that the optimization problems defined and encoded by ADORE and the ADSG perform as well as or better than manually-formulated optimization problems.

How to solve SAO problems using global optimization algorithms? By using global optimization algorithms specifically adapted for SAO. Evolutionary algorithms such as NSGA-II can deal with most of the SAO optimization challenges, except that they require a large number of function evaluations to converge, which is problematic if evaluation is expensive. For expensive evaluation, Bayesian Optimization (BO) algorithms are more appropriate as they use Gaussian Process (GP) models to create a surrogate model of the design space and thereby reach the optimum (or Pareto front in case of multiple objectives) in less function evaluations. These global optimization algorithms were adapted for SAO by:

- developing a hierarchical sampling algorithm that ensures regions of the design space are not over- or under-represented, thereby considering max rate diversity (MRD) effects;
- investigating how correction ratio (CR) effects can be considered by comparing several problem-agnostic correction algorithms to problem-specific correction, showing that problem-specific greedy correction is sufficient;
- integrating information about the hierarchical design space into the optimization algorithm to support generating valid (corrected and imputed) design vectors and to leverage activeness information for sampling and model creation, thereby considering imputation ratio (IR) and correction ratio (CR) effects; and
- developing a strategy for handling hidden constraints in BO, by training a model to predict the Probability of Viability (PoV) and including that model in the infill search process as an inequality constraint.

The adaptations were tested using three benchmark problems and a jet engine architecture optimization problem that features hidden constraints. It is shown that BO can solve SAO problems with up to 92% less function evaluations than NSGA-II, showing it is appropriate for solving expensive SAO problems. Algorithms and test problems are implemented in the open-source SBArchOpt library[2] as a result of this work.

How to evaluate and optimize system architecture instances in a collaborative MDAO environment? By enabling dynamically changing MDAO workflow behavior, propagating architecture instance data to the MDAO workflow, and running the architecture generator in the same environment used for running the MDAO workflow:

- Dynamic behavior in collaborative MDAO workflows was enabled by identifying architectural influences (discipline activation, discipline repetition, data connections, and conditional variables) and defining several high-level strategies for

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

implementing dynamic MDAO: the "single dynamic" strategy was applied, where a single MDAO workflow is deployed that dynamically changes its behavior during execution.

Then, methods were developed to define architectural influences in MDAO workflows (activation assertions, repetition iterators, and global-to-local mapping), and finally the methods were implemented in MDAx (a DLR in-house MDAO workflow modeling tool) and its RCE (Remote Component Environment; a Process Integration and Design Optimization (PIDO) platform) workflow export.

- Architecture instance data was propagated from ADORE to the Central Data Schema (CDS) using the Node Factory Evaluator (NFE), which defines rules to dynamically create XML nodes (for an XML-based CDS) for associated architecture elements, and metric factories to read resulting performance values from the CDS and associate them to output metrics (used as objectives and constraints).
- Architecture generation was integrated in the collaborative MDAO environment by combining the ADORE architecture generator, the optimization algorithm, and the NFE as a tell-ask block in RCE, and allowing the user to configure the optimization problem and algorithm from the ADORE GUI.

These techniques were demonstrated by a multi-stage launch vehicle SAO problem, showing the development of the dynamic collaborative MDAO workflow and a demonstration of the NFE for propagating data from the generated ADORE architectures to the CDS, with integrated execution of the architecture generation step in the RCE workflow.

5.2. RECOMMENDATIONS

Here, the author would like to recommend some research topics and courses of action to move research into and application of SAO forwards.

5.2.1. SYSTEM ARCHITECTURE OPTIMIZATION METHODOLOGY IMPLEMENTATION (DSG, ADSG AND ADORE)

- Function fulfillment choice options (ADSG) are mutually-exclusive, and modeling options for multiple components fulfilling a function together is possible using multi-fulfillment elements. More elaborate *cardinality constraints* [94] might increase expressivity of such selection choices. Care should be taken to not increase modeling complexity by too much.
- The previous recommendation might be supported by also *allowing connection choices to influence node selection* in the DSG, and allowing chaining connection and selection choices in any order (compared to currently only resolving connection choices after resolving selection choices).

  For example, target nodes that are not connected after resolving a connection choice might be removed from the DSG, having the same impact as a removed selection choice option node. This would allow using connection choices to model function fulfillment choices, and in general allow supporting more general architecture pattern chaining, as for example applied by APAZA & SELVA [122].

GHANJAOUI ET AL. presented a DSG application case about co-development of cabin and assembly architectures where this functionality would have been useful [29]: connection interfaces (between cabin elements and the airframe) were assigned to joining technologies using connection choices, however the assignment of joining technologies had downstream influence on the selection of resources for performing the associated assembly process steps (modeled using selection choices). Since connection choices do not influence downstream selection choices, part of the architecture design space was redundant.

- The complete selection choice encoder of the DSG should be extended with the capability to detect sets of selection choices where at most one selection choice is active at a time. This can for example occur if there are selection choices that are derived by separate option nodes of an upstream selection choice. Detecting such sets allows to *collapse* the selection choices in the set into one design variable, reducing the discrete imputation ratio IR_d.
- The complete selection choice encoder of the DSG becomes inefficient (in time and memory usage) from a certain design space size. To support larger design spaces, the *complete encoder should be improved*, for example by applying Constraint Satisfaction Problem (CSP) [266] solving techniques or Satisfiability Modulo Theory (SMT) [267].

ARCHITECTURE DESIGN SPACE MODELING METHODOLOGY

- Better support should be implemented for *progressively decomposing* an architecture design space model, which would facilitate progressively refining the system architecture as suggested by [39, 268]. To enable this, it should for example be possible to convert a component to a subsystem block, and having that system maintain its functional interfaces (fulfilled and needed functions).
- Modeling variability in *functional architecture* (system behavior) should be supported. This is relevant when it is possible to consider different concepts of operation for a system [34, 39], and is especially relevant in the context of System-of-Systems [269].

  To support this, it should first be possible to model functional architecture in more details, mainly supporting function flow (inputs and outputs) to model connections and dependencies between functions [62, 270]. Then, variability can be modeled using selection choices to select alternative preceding or subsequent functions. It is expected that the bottom-up approach (see previous recommendation) works best for that, which should be verified through further investigation.
- Architecture generation and evaluation are currently strictly separated, however allowing some level of evaluation already in the architecture generation environment might help speed up the process by *filtering-out architectures* with a low chance of being optimal. This could be achieved by supporting some level of simple calculations, for example mass or Technology Readiness Level (TRL) aggregation [271, 272], or by calculating metrics directly from the architecture itself, for example complexity [102, 273] or decomposability [274].

- Currently, the complete architecture design space is specified in one ADORE model, however to increase reusability and improve modularity it should be possible to *compose architecture design spaces* from multiple models.

  One way would be to make it possible to import other ADORE models as a system block, and in the other direction to extract system blocks as separate ADORE models. Some mechanism should be implemented to keep function definitions synchronized between models, as for example a function in the top-level system would be a boundary function in the extracted model of a subsystem. These functionalities would be useful for application cases like System-of-Systems and simultaneous development of the system and its enabling systems (i.e. manufacturing and maintenance systems).

MBSE Process Integration

- Because ADORE does not represent a full Architecture Description Language (ADL) [275] (e.g. because stakeholder, needs, and various other contextual elements are missing), connections should be established between ADORE and existing *standardized MBSE languages*, such as SysML [55], OPM [57], and Arcadia/Capella [59].

  An especially promising candidate is SysMLv2 [89], as it defines standardized APIs for data integration and includes variability modeling as an integral part of the language. For example, ADORE functions can be mapped to SysMLv2 actions, components to parts, selection choices to variation definitions, design problems to trade studies, and generated architectures to trade study alternatives.

- Semantic and traceable connections should be established from *requirements* to architecture elements, including but not limited to (boundary) functions defined from functional requirements and QOIs defined from performance requirements.

- It should be possible to run *architecture-level sensitivity analysis* studies, to investigate what the impact is of inputs other than architectural choices or design variables on optimization results. For example, to investigate the impact of assumptions (such as battery energy density, or the price of something), or to investigate the impact of exchanging some technology for another.

  This sensitivity analysis process can be automated by running the SAO loop nested within a DoE over such "assumption scenarios". This process might be accelerated by reusing evaluation results across scenarios, for example using multi-fidelity surrogate modeling techniques.

- It should be investigated whether *generative Artificial Intelligence (AI)* [276, 277] techniques can be used to support and/or accelerate certain steps in the system architecting process. Generate AI may be used to improve the system architecting process by:

  - supporting the formulation of the design space by suggesting technology/component options for fulfilling a given function, identifying incompatibilities or potential synergies, and/or decomposing higher-level functions into more detailed lower-level functions;

  - helping engineers that are not familiar with the modeling tool with modeling architecture design spaces;
  - supporting design space exploration by suggesting analysis models to be used or by suggesting interesting architecture instances to consider (e.g. for an initial sampling or as a goal in itself); and/or
  - supporting decision-making by answering questions about design space exploration results to help the systems engineer understand the design space, for example as demonstrated by APAZA & SELVA [278].

OTHER APPLICATION DOMAINS

- It should be investigated whether SAO principles are applicable to *Model-Based Product Line Engineering (MBPLE)*. PLE originated to manage product lines, for managing configurations of a system where for each application of that system (e.g. different customers) a different configuration is appropriate. However, recent efforts have included MBPLE in system design phases to define architecture instances, most notably in Airbus' MOFLT (Mission - Operation - Function - Logical - Technical) framework [97].
- SAO can be made applicable to more engineering domains by supporting the creation of *domain-specific architecture design space modeling languages*. Just as the ADSG and ADORE can be seen as layers on top of the DSG specific to the system architecting domain, other domains could be supported by similar mechanisms.

  For that, the mechanisms for defining new node types (for example Table 3.6) and choice definition rules currently implemented in the ADSG and the definition of the associated GUI (ADORE) should be generalized. Possible application domains include aircraft onboard systems [42, 106, 279, 280], System-of-Systems [269, 281], and manufacturing systems [282].

  Other applications may include formulating hyperparameter tuning problems for automated machine learning (AutoML) [135, 230], neural network architecture search problems [283, 284], and formulating hierarchical design spaces in general [26, 188]. These mechanisms can also be used to integrate with standardized MBSE languages (e.g. SysMLv2), by enabling architecture design space modeling using the same terminology as the connected language.
- The previous recommendation can also be (partly) implemented by supporting the development of Domain-Specific Languages (DSLs) [285]: programming languages that are developed for a specific application domain (as opposed to general-purpose programming languages). First, a general *SAO-DSL* for defining DSGs and/or ADSGs can be developed. Compared to the Python API for defining DSGs in ADSG CORE, this might already offer a more intuitive text-based interface for defining DSGs. This general SAO-DSL can then be used as a basis for developing DSLs for specific application domains (see previous recommendation).

5.2.2. Optimization Algorithms

Current Methods

- Including more *information about the hierarchical design space* generally improves optimizer performance (see Section 2.3.4), however for some specific cases this is not true.

  For NSGA-II, for example, the single-objective versions of the GNC problem performed better if no hierarchy information was included (the Naive approach), see Table 2.12. For BO, the same test problems performed as well as with full hierarchy information availability, see Table 2.13, whereas a worse performance would be expected.

  It should be investigated what problem properties can lead to this behavior, and then either correct for these properties or dynamically adapt the hierarchy integration strategy based on such properties.

- Optimization performance of the BO algorithm should be improved for:

  - Problems with *many design variables* (i.e. up to hundreds [286]), as currently GP models have difficulty training to many dimensions, or need too much time for practical application of BO in high-dimensional design spaces.

    Techniques to investigate include dimension reduction techniques such as Kriging with Partial Least Squares (KPLS) [287], Manifold GPs [288], and EGO with Random and Supervised Embedding (EGORSE) [185]. However, evidence exists that normal GPs can be used for high-dimensional design spaces too [289].
  - Problems with *many objectives*, as in this work only problems with up to 2 objectives have been tested. Examples of research in this area include [290, 291].

- The hierarchical sampling algorithm presented in Section 2.3.2 depends on the availability of $x_{\text{valid,discr}}$, which might not be available due to memory or time limits for very large design spaces. A hierarchical sampling algorithm should be developed that *does not depend on the availability of* $x_{\text{valid,discr}}$.

- An extension to *multi-fidelity BO* [292] should be considered to speed up BO processes if it is possible to formulate architecture evaluation functions at different levels of fidelity. This might combine well with the recommendation to allow defining simple calculation in the architecture design space model.

- Automatically *decomposing* SAO problems into multiple nested optimization problems [139] should be investigated, for example to benefit from the availability of gradients for continuous variables. The application case presented in Section 3.5 was implemented as a multi-level optimization problem, albeit manually.

NEW METHODS

- Considering uncertainty in the architecture evaluation is important for supporting the decision-making process [293]. It should therefore be possible to *propagate uncertainty* throughout the SAO loop [294]. This also enables the application of robust optimization techniques [49], with the result of finding (Pareto) optimal architectures that are robust in changes in the system context and/or to evaluation model assumptions.
- The GP models in BO algorithms for SAO are trained using distance metrics applied to the design vectors, and even though the selection and connection choice encoders are developed such that a small change in a design vector should result in a small change in the associated system architecture (i.e. the encoders should exhibit non-degradedness, see Section 3.1.2), this agreement will not be perfect in all cases.

  It should be investigated whether it is also possible directly train the GP models on differences in system architecture, represented by difference in their DSGs, which can then be used to define *graph kernels* [295–299].
- Alternatively to GP models using graph kernels, it should be investigated whether *graph neural networks* [300–302], which directly learn performance metrics from the graph structure, are also applicable. It should be investigated whether including more generic output metrics (instead of only the objective and constraint values to be predicted) improves predictor performance.
- Solving the MDAO problem should be made more efficient, and a good candidate for achieving this is *Efficient Global MDO (EGMDO)* [303]. Recently, EGMDO has been extended to support design constraints [304], however for application in SAO it should also support mixed-discrete variables, multiple objectives, hidden constraints, all architecture influences as identified in this work, and it should be compatible with collaborative MDAO techniques.

5.2.3. ARCHITECTURE EVALUATION AND MDAO

COLLABORATIVE AND DYNAMIC MDAO

- The architecture generator step (as implemented in the tell-ask block) should be included in MDAx dynamic MDAO workflow models as an *optimizer block*, with appropriate inputs and outputs as translated by the NFE.
- *CMDOWS* [264] should be extended to support all identified architectural influences in dynamic MDAO, building on partial support for architectural influences as presented in [213, 214].
- Support for dynamic MDAO should be implemented in *more MDAO frameworks*, such as OpenMDAO [248], GEMSEO [305], CoSApp [306], CSDL [307], and commercial PIDO environments. To achieve this, a common standard format should be developed (or an existing standard should be extended) that specifies such dynamic MDAO behavior, which can then be implemented by the various platforms.

- A consistent extension of the *XDSM* notation should be developed, that supports visualizing dynamic MDAO behavior relevant to the MDAO research community. The first contributions of Chapter 4 and Figures 8 and 9 of [213] should be considered.
- It should be investigated whether it is possible to generate and execute collaborative MDAO workflow *on-demand* (see Figure 4.8), by automating the setup steps as currently still required in the PIDO environment. This would effectively provide an advanced optimization MDAO bot as proposed in Figure 8.7b in [208].

Architecture Evaluation Methods

- The relationship between the architecture design space and the *simulation architecture* [197] should be investigated more, and guidelines should be developed to support the implementation of architecture evaluation for SAO. Special focus should be placed on making sure that the simulation architecture is flexible enough to support changing its structure and behavior for different architecture instances. The development of the design space model and simulation architecture should be more formally linked by (not exhaustive):
  - Defining the simulation architecture based on the system architecture design space. Preliminary work on applying MBSE to the formulation of MDAO systems can be used as a basis, such as [203, 308].
  - Automatically generating architecture influence logic for dynamic MDAO from the architecture design space.
  - Analyzing whether all parts of the architecture design space are sufficiently covered by the simulation architecture.
  - Generating a library of components that can be used in defining the architecture design space from the library of simulation blocks available in a simulation environment [309].

    This can be used to connect to component-based simulation tools like Simulink [3], Hopsan [310], PACE SysArc [311], or GTlab [312]. Halsema [294] used a component library for defining electric heavy-duty vehicle drivetrain architecture design spaces.
- It should be investigated how *Knowledge Based Engineering (KBE)* methods can be applied for SAO. KBE should be suitable for SAO, as it relies on object-oriented construction of system-level models using low-level primitives, and fundamentally treats these as subject to multidisciplinary analysis [71]. The object-oriented nature should make it integrate well with the CFE (see Table 3.7). It also ensures that the KBE model supports the types of architectural changes present in SAO design spaces, such as component selection (replacement) and component instantiation.

[3]https://www.mathworks.com/help/simulink/block-libraries.html

5.3. OUTLOOK

Only a relatively small, although important, set of aspects related to SAO could be explored in the scope of this dissertation. In this final section, the author would like to share a vision for the future of complex systems engineering where SAO plays an important role. In that future . . .

SAO is a well-known and well-established method in the systems engineer's toolbox, that complements and integrates with existing MBSE methods, and is applied in industry contexts for the architecting phases of many complex systems and in particular novel systems with a need for simulation-based evaluation. Defining SAO problems can be done with little additional training, by being integrated in standardized MBSE languages. SAO problems can be executed without switching software environments and without the need for writing code (except as needed for architecture evaluation). Ultra-efficient global optimization algorithms exist that can solve SAO problems with very large, highly-constrained design spaces with many conflicting objectives within a low number of architecture and/or disciplinary tool evaluations. The appropriate optimization algorithm is automatically selected based on SAO problem characteristics. It is easy to develop and deploy multidisciplinary analysis toolchains, with disciplines, MDAO strategy, and dynamic workflow behavior tightly coupled with the architecture design space model, and with a focus on leveraging collaborative MDAO and Knowledge-Based Engineering (KBE) principles.

Uncertainties in evaluation models and/or assumptions in input parameters are propagated throughout the complete SAO loop, and are used for decision-making and robust optimization. Resulting Pareto-optimal architectures are available in the MBSE environment, and are the basis for subsequent detailed design phases without much (if any) additional work. Architecture design space and evaluation models are reusable between SAO campaigns within a project and between projects.

An interdisciplinary and international research community has been established around the SAO topic, that tightly integrates with the systems engineering, optimization, and MDAO communities. As part of the research community, regular meetings are organized for knowledge exchange and collaboration, SAO is applied as core methodology in various research projects featuring elaborate industry-backed application cases, and training material is available for audiences with various levels of expertise. In the systems engineering community, SAO is seen as an important enabler for the system architecting process, and knowledge exchange with the Product Line Engineering (PLE) and variability modeling communities is established. In the MDAO community, SAO is seen as a relevant method for defining design space exploration problems, and dynamic MDAO is seen as an important specialization of MDAO, with rigorous theory developed about the mutual influence of architecture design space models and MDAO workflows, and with a standardized notation for dynamic MDAO workflows.

It is easy to adopt SAO for domain-specific applications, by defining domain-specific ontology and rules, and by defining the design space model based on analysis models available in domain-specific analysis libraries. The application of SAO has expanded to other application cases such as the design of aircraft onboard systems and System-of-Systems, and to other domains such as space, defense, energy, and infrastructure.

ACKNOWLEDGEMENTS

This dissertation has been the result of six years of research performed at the Institute of System Architectures in Aeronautics of the German Aerospace Center (DLR) in Hamburg, Germany, in collaboration with many friends and colleagues.

First of all, I would like to thank Pier Davide Ciampa and Björn Nagel for allowing me to work with you on establishing digital methods for designing complex, multidisciplinary aviation systems. Thank you Pier Davide for teaching me about collaborative MDAO, for involving me in interesting research projects and in particular AGILE 4.0, for giving me the freedom to try out whether we could "automatically generate system architectures", for encouraging me to keep the bigger picture (the "system-level picture") in mind, and for guiding me in the first steps of my professional development. Thank you Björn for supporting me in my research, for enabling me to spend time exploring new topics, for supporting and encouraging me to visit international conferences, and for supporting and enabling my three-month secondment at ONERA in Toulouse.

I want to thank all my current and former colleagues and friends at the DLR institute in Hamburg that have made and keep making it such a pleasant work environment, in particular but not limited to: Luca for supporting me in my research and my secondment at ONERA, Andreas for the many interesting discussions about MDAO and ontologies, and for developing MDAx with me, Pina for putting ADORE to the test in your dissertation and by implementing the largest design space to date (the number of declared supply chain architectures was about 5e36!), Francesco for the interesting discussions on development processes, digital engineering and architecture frameworks, Sparsh for joining me in applying and further developing MDAx, and for continuing the research into dynamic MDAO in the context of SAO, Adrian for giving ADORE a try in the AGILE 4.0 Academy and later seeing how we can connect requirements to system architecture, Carlos for, after being the first beta-tester of the ADORE GUI, still being bug-finder number one and demonstrating the potential of SAO for designing aircraft systems in your PhD research, Jasamin for expanding ADORE to the world of System-of-Systems engineering in your master's thesis and now in the COLOSSUS project, Raúl for, still as a student, preparing and running the hybrid-electric propulsion system optimization problem, for further developing the theory behind dynamic MDAO, providing the first implementation of it in MDAx, and for creating the rocket architecture problem, Nabih and Naz for optimizing SoS architectures in COLOSSUS, Prajwal for always coming up with new ways to collaborate and spread our methods, Yassine for applying the DSG to your assembly process optimization framework, and Erwin and Marko for the interesting discussions on digital threads, and for planning the next DLR-wide digital engineering project featuring SAO and MDAO with me.

On the academic side, I would like to thank my promotor Gianfranco La Rocca in Delft and my copromotor Nathalie Bartoli in Toulouse. Thank you Gianfranco for seeing the potential of SAO for designing complex multidisciplinary systems, guiding me in topic scoping and selecting research questions, asking tough questions to drive and steer the research, reviewing my works, the good discussions on MDAO and optimization, and for co-supervising our students. Thank you Nathalie for contributing and providing feedback to our papers on optimization and my research work in general, introducing me to Bayesian Optimization, and inviting me to Toulouse for a deep dive into such optimization algorithms.

In addition to Nathalie, I received a very warm welcome at the DTIS department of ONERA in Toulouse from Paul, Thierry, Rémi, Joseph, Sebastien, Eric, Christophe, Inês, Sam, Charles, Rémy, Jason, Anouck, Luiz, Romain, Romain, Hugo, Camille, and Julien. Thank you all for making my three months there an unforgettable experience, and teaching me how to play Tarot (unfortunately I only ever won the "ears"...). In particular I would like to thank Nathalie, Paul, Thierry, and Rémi for letting me work on optimization algorithms with you, seeing the value of explicitly modeling hierarchical design spaces, asking tough questions about my work, and providing valuable feedback. Thank you Paul in particular, for the many interesting discussions on Gaussian Processes (I still only have a basic understanding compared to you!) and Bayesian Optimization, and for involving me in the development of SMT 2.0.

During my research I had the pleasure of supervising several students. Next to Carlos, Jasamin and Raúl, thank you Thibault for taking on the challenge of developing the jet engine benchmark with me, and Mahmoud for developing the hybrid-electric propulsion architecture simulation framework with me, and for connecting our research to the MDO lab in Michigan. Thank you all for teaching me how to guide and challenge you, for forcing me to stay humble, for the great work delivered, and for the great time working and discussing with you. In this context I would also like to thank Daniël, for letting Raúl and me pick your mind about rockets and launch trajectories, and for together with Sarah making my conferences in the US more fun.

In addition to the above, I would like to thank everyone else who helped me in my work by applying and testing my methods and tools, and for publishing with me: thank you Thomas, Eytan, Joaquim, Nils, Andrew, Mona, Susan, Sami, Marco, Anne-Liza, Santiago, Luuk, Feliciano, Pasquale, David, Jd, Nehi, and all the colleagues in AGILE 4.0, COLOSSUS, MBSE-Ops, and other projects.

Finally, I would like to thank all my family and friends for supporting me during these years. Thank you Mama, Papa, and Mathijs for making me who I am, supporting me through thick and thin, sparking my interests in programming and engineering (doesn't matter if it floats or flies), and encouraging me to stay curious and follow my dreams, even if abroad. Finally there is the love of my life, the reason I returned to Hamburg: my wife Melissa. Thank you for being who you are, for sharing my passion for aviation, for showing me Italy and Toronto and exploring other places with me, for supporting me whenever I find bugs, for encouraging me to go to Toulouse and for trying out all the restaurants there with me, and for building a life with me in Hamburg. I can't wait to see what our future with our son Carlo and cats Elmi and Jip will bring!

Jasper Bussemaker
Hamburg, May 2025

References

See the List of Publications for publications involving the author of this dissertation.

[31] D.S. Lee et al. "The contribution of global aviation to anthropogenic climate forcing for 2000 to 2018". In: *Atmospheric Environment* 244 (Jan. 2021), p. 117834. DOI: 10.1016/j.atmosenv.2020.117834.

[32] A. Barke et al. "Are Sustainable Aviation Fuels a Viable Option for Decarbonizing Air Transport in Europe? An Environmental and Economic Sustainability Assessment". In: *Applied Sciences* 12.2 (Jan. 2022), p. 597. DOI: 10.3390/app12020597.

[33] E.J. Adler and J.R.R.A. Martins. "Hydrogen-powered aircraft: Fundamental concepts, key technologies, and environmental impacts". In: *Progress in Aerospace Sciences* 141 (Aug. 2023), p. 100922. DOI: 10.1016/j.paerosci.2023.100922.

[34] INCOSE. *INCOSE Systems engineering handbook*. Fifth edition. Hoboken, NJ: Wiley., 2023. 1344 pp. ISBN: 978-1-11981-429-0.

[35] D. Kellari, E.F. Crawley, and B.G. Cameron. "Architectural Decisions in Commercial Aircraft from the DC-3 to the 787". In: *Journal of Aircraft* 55.2 (Mar. 2018), pp. 792–804. DOI: 10.2514/1.C034130.

[36] P.D. Ciampa and B. Nagel. "AGILE Paradigm: The next generation of collaborative MDO for the development of aeronautical systems". In: *Progress in Aerospace Sciences* 119 (Nov. 2020). DOI: 10.1016/j.paerosci.2020.100643.

[37] F. Afonso et al. "Strategies towards a more sustainable aviation: A systematic review". In: *Progress in Aerospace Sciences* 137 (Feb. 2023), p. 100878. DOI: 10.1016/j.paerosci.2022.100878.

[38] NASA. *NASA Systems Engineering Handbook*. Tech. rep. Rev 2. NASA, 2016, p. 356.

[39] E. Crawley, B. Cameron, and D. Selva. *System architecture: strategy and product development for complex systems*. England: Pearson Education, 2015. ISBN: 978-0-1339-7534-5.

[40] K.A. Salem, G. Palaia, and A.A. Quarta. "Review of hybrid-electric aircraft technologies and designs: Critical analysis and novel solutions". In: *Progress in Aerospace Sciences* 141 (Aug. 2023), p. 100924. DOI: 10.1016/j.paerosci.2023.100924.

[41] M.F. Shahriar and A. Khanal. "The current techno-economic, environmental, policy status and perspectives of sustainable aviation fuel (SAF)". In: *Fuel* 325 (Oct. 2022), p. 124905. DOI: 10.1016/j.fuel.2022.124905.

[42] R. Bornholdt, T. Kreitz, and F. Thielecke. “Function-Driven Design and Evaluation of Innovative Flight Controls and Power System Architectures”. In: *SAE Technical Paper Series*. SAE International, Sept. 2015. DOI: 10.4271/2015-01-2482.

[43] K.S. Ng, D. Farooq, and A. Yang. “Global biorenewable development strategies for sustainable aviation fuel production”. In: *Renewable and Sustainable Energy Reviews* 150 (Oct. 2021), p. 111502. DOI: 10.1016/j.rser.2021.111502.

[44] L. Martinez-Valencia, M. Garcia-Perez, and M.P. Wolcott. “Supply chain configuration of sustainable aviation fuel: Review, challenges, and pathways for including environmental and social benefits”. In: *Renewable and Sustainable Energy Reviews* 152 (Dec. 2021), p. 111680. DOI: 10.1016/j.rser.2021.111680.

[45] D. Silberhorn et al. “The Air-Vehicle as a Complex System of Air Transport Energy Systems”. In: *AIAA AVIATION 2020 FORUM*. American Institute of Aeronautics and Astronautics, June 2020. DOI: 10.2514/6.2020-2622.

[46] J.V. Iacobucci. “Rapid Architecture Alternative Modeling (Raam): a Framework for Capability-Based Analysis of System of Systems Architectures”. PhD thesis. Georgia Institute of Technology, 2012.

[47] R. Haberfellner et al. *Systems Engineering*. Cham: Springer International Publishing, 2019. DOI: 10.1007/978-3-030-13431-0.

[48] T.A. McDermott, D.J. Folds, and L. Hallo. “Addressing Cognitive Bias in Systems Engineering Teams”. In: *30th Annual INCOSE International Symposium*. Virtual Event, July 2020. DOI: 10.1002/j.2334-5837.2020.00721.x.

[49] J.R.R.A. Martins and A. Ning. *Engineering Design Optimization*. Cambridge, UK: Cambridge University Press, Jan. 2022. DOI: 10.1017/9781108980647.

[50] P. Shiva Prakasha et al. “COLOSSUS EU Project - Collaborative SoS Exploration of Aviation Products, Services and Business Models: Overview and Approach”. In: *34th Congress of the International Council of the Aeronautical Sciences, ICAS 2024*. Florence, Italy, Sept. 2024.

[51] ISO/IEC/IEEE. *International Standard 15288-2015 - Systems and software engineering – System life cycle processes*. DOI: 10.1109/ieeestd.2015.7106435.

[52] A.M. Madni and M. Sievers. “Model-Based Systems Engineering: Motivation, Current Status, and Needed Advances”. In: *Disciplinary Convergence in Systems Engineering Research*. Springer International Publishing, Nov. 2017, pp. 311–325. DOI: 10.1007/978-3-319-62217-0_22.

[53] J. Chaudemar and P. de Saqui-Sannes. “MBSE and MDAO for Early Validation of Design Decisions: a Bibliography Survey”. In: IEEE, Apr. 2021. DOI: 10.1109/syscon48628.2021.9447140.

[54] J. Holt and S. Perry. *SysML for systems engineering: 2nd edition: A model-based approach*. Institution of Engineering and Technology, Nov. 2013, pp. 1–935. DOI: 10.1049/PBPC010E.

[55] S. Friedenthal, A. Moore, and R. Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Vol. 38. 1/2. Elsevier, 2015. DOI: 10.1016/C2013-0-14457-1.

[56] B. Bagdatli et al. “SysML State of the Art”. In: *Handbook of Model-Based Systems Engineering*. Springer International Publishing, 2022, pp. 1–31. DOI: 10.1007/978-3-030-27486-3_13-1.

[57] D. Dori. *Model-based systems engineering with OPM and SysML*. Springer New York, 2016, pp. 1–411. DOI: 10.1007/978-1-4939-3295-5.

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

[59] J. Voirin, ed. *Model-based system and architecture engineering with the arcadia method*. London: ISTE Press, 2018. ISBN: 978-1-78548-169-7.

[60] P. Roques, ed. *Systems architecture modeling with the Arcadia method. A practical guide to Capella*. Implementation of model based system engineering set. London: ISTE Press Ltd, 2018. DOI: 10.1016/C2016-0-00854-9.

[61] D. Mavris, C. de Tenorio, and M. Armstrong. “Methodology for Aircraft System Architecture Definition”. In: *46th AIAA Aerospace Sciences Meeting and Exhibit*. January. Reston, Virigina: American Institute of Aeronautics and Astronautics, Jan. 2008, pp. 1–14. DOI: 10.2514/6.2008-149.

[62] T. Weilkiens, J.G. Lamm, and S. Roth. *Model-Based System Architecture*. Hoboken, NJ, USA: Wiley John and Sons, Nov. 2, 2015. 400 pp. DOI: 10.1002/9781119051930.

[63] Y. Menshenin et al. “Model-Based System Architecting and Decision-Making”. In: *Handbook of Model-Based Systems Engineering*. Cham: Springer International Publishing, 2022, pp. 1–42. DOI: 10.1007/978-3-030-27486-3_17-1.

[64] J. Hirtz et al. “A functional basis for engineering design: Reconciling and evolving previous efforts”. In: *Research in Engineering Design* 13.2 (Mar. 2002), pp. 65–82. DOI: 10.1007/s00163-001-0008-3.

[65] D. Horber et al. “Utilizing System Models for Multicriteria Decision-Making—A Systematic Literature Review on the Current State of the Art”. In: *IEEE Open Journal of Systems Engineering* 2 (2024), pp. 135–147. DOI: 10.1109/ojse.2024.3434310.

[66] R. Razzouk and V. Shute. “What Is Design Thinking and Why Is It Important?” In: *Review of Educational Research* 82.3 (Sept. 2012), pp. 330–348. DOI: 10.3102/0034654312457429.

[67] D.J. Singer, N. Doerry, and M.E. Buckley. “What Is Set-Based Design?” In: *Naval Engineers Journal* 121.4 (Oct. 2009), pp. 31–43. DOI: 10.1111/j.1559-3584.2009.00226.x.

[68] N. Shallcross et al. “Set-based design: The state-of-practice and research opportunities”. In: *Systems Engineering* 23.5 (July 2020), pp. 557–578. DOI: 10.1002/sys.21549.

[69] K. Czarnecki et al. “Cool features and tough decisions”. In: *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems - VaMoS ’12*. Leipzig, Germany: ACM Press, 2012. DOI: 10.1145/2110147.2110167.

[70] S. Gedell and H. Johannesson. “Design rationale and system description aspects in product platform design: Focusing reuse in the design lifecycle phase”. In: *Concurrent Engineering* 21.1 (Dec. 2012), pp. 39–53. DOI: 10.1177/1063293x12469216.

[71] G. La Rocca. “Knowledge Based Engineering Techniques to Support Aircraft Design and Optimization”. PhD thesis. Delft University of Technology, 2011, p. 313. ISBN: 978-909026069-3.

[72] J. Sobieszczanski-Sobieski, A. Morris, and M.J.L. van Tooren. *Multidisciplinary Design Optimization Supported by Knowledge Based Engineering*. West Sussex, UK: John Wiley & Sons, Ltd, Aug. 2015, pp. 1–378. DOI: 10.1002/9781118897072.

[73] E. Silvas et al. “Review of Optimization Strategies for System-Level Design in Hybrid Electric Vehicles”. In: *IEEE Transactions on Vehicular Technology* (2016), pp. 1–1. DOI: 10.1109/tvt.2016.2547897.

[74] M.N. Roelofs and R. Vos. “Correction: Uncertainty-Based Design Optimization and Technology Evaluation: A Review”. In: *2018 AIAA Aerospace Sciences Meeting*. Reston, Virginia, Jan. 2018. DOI: 10.2514/6.2018-2029.c1.

[75] IEEE/ISO/IEC. *International Standard 42020-2019 - Software, systems and enterprise – Architecture processes*. IEEE, 2019. DOI: 10.1109/ieeestd.2019.8767004.

[76] S. Procter and L. Wrage. “Guided Architecture Trade Space Exploration: Fusing Model-based Engineering and Design by Shopping”. In: *Software and Systems Modeling* 20.6 (June 2021), pp. 2023–2045. DOI: 10.1007/s10270-021-00889-8.

[77] C. Habermehl et al. “Optimization Workflows for Linking Model-Based Systems Engineering (MBSE) and Multidisciplinary Analysis and Optimization (MDAO)”. In: *Applied Sciences* 12.11 (May 2022), p. 5316. DOI: 10.3390/app12115316.

[78] A. Bruggeman and G. La Rocca. “From Requirements to Product: an MBSE Approach for the Digitalization of the Aircraft Design Process”. In: *INCOSE International Symposium* 33.1 (July 2023), pp. 1688–1706. DOI: 10.1002/iis2.13107.

[79] D.M. Judt and C.P. Lawson. “Development of an automated aircraft subsystem architecture generation and analysis tool”. In: *Engineering Computations* 33.5 (July 2016), pp. 1327–1352. DOI: 10.1108/EC-02-2014-0033.

[80] D. Selva. “Rule-based system architecting of Earth observation satellite systems”. PhD thesis. Massachusetts Institute of Technology, Dept. of Aeronautics and Astronautics, 2012.

[81] A.M. Madni. “Novel Options Generation”. In: *Transdisciplinary Systems Engineering*. Cham: Springer International Publishing, Oct. 2017, pp. 89–102. DOI: 10.1007/978-3-319-62184-5_6.

[82] A.M. Madni and S. Purohit. “Economic Analysis of Model-Based Systems Engineering”. In: *Systems* 7.1 (Feb. 2019), p. 12. DOI: 10.3390/systems7010012.

[83] M. Svahnberg, J. van Gurp, and J. Bosch. “A taxonomy of variability realization techniques”. In: *Software: Practice and Experience* 35.8 (2005), pp. 705–754. DOI: 10.1002/spe.652.

[84] H. Broodney et al. “Generic Approach for Systems Design Optimization in MBSE”. In: *INCOSE International Symposium* 22.1 (July 2012), pp. 184–200. DOI: 10.1002/j.2334-5837.2012.tb01330.x.

[85] P. Leserf, P. de Saqui-Sannes, and J. Hugues. “Trade-off analysis for SysML models using decision points and CSPs”. In: *Software and Systems Modeling* 18.6 (Jan. 2019), pp. 3265–3281. DOI: 10.1007/s10270-019-00717-0.

[86] B. Mas Sanz et al. “Lean Variation Management for Satellite Architectures: A SysML Approach”. In: *MBSE2024 Workshop*. Bremen, Germany, May 2024.

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

[88] A.A. Kerzhner. “Using logic-based approaches to explore system architectures for systems engineering”. PhD thesis. Georgia Institute of Technology, 2012.

[89] M. Bajaj, S. Friedenthal, and E. Seidewitz. “Systems Modeling Language (SysML v2) Support for Digital Engineering”. In: *INSIGHT* 25.1 (Mar. 2022), pp. 19–24. DOI: 10.1002/inst.12367.

[90] T. Weilkiens and C. Muggeo. *Don’t panic! The absolute beginner’s guide to SysML v2*. Ed. by S. Perry. Ilminster, Somerset: INCOSE UK, 2023. 59 pp. ISBN: 9781739463106.

[91] L. Timperley et al. “Mapping the MBSE Environment and Complementary Design Space Exploration Techniques”. In: *2024 IEEE Aerospace Conference*. IEEE, Mar. 2024, pp. 1–20. DOI: 10.1109/aero58975.2024.10521188.

[92] S. Friedenthal. *Introduction to the SysML v2 Language Graphical Notation*. Tech. rep. Mar. 2023.

[93] H. Wagner, W. Rössler, and C. Zuccaro. “How to Structure Variability in Large-Scale Complex Engineered Systems”. In: *Tag des Systems Engineering*. Würzburg, Germany, Nov. 2023.

[94] C. Quinton, D. Romero, and L. Duchien. “Cardinality-based feature models with constraints: a pragmatic approach”. In: *Proceedings of the 17th International Software Product Line Conference*. SPLC 2013. ACM, Aug. 2013. DOI: 10.1145/2491627.2491638.

[95] J. Meinicke et al. *Mastering Software Variability with FeatureIDE*. Cham: Springer International Publishing, 2017. DOI: 10.1007/978-3-319-61443-4.

[96] S.A. Safdar et al. "Evaluating Variability Modeling Techniques for Supporting Cyber-Physical System Product Line Engineering". In: *Lecture Notes in Computer Science*. Cham: Springer International Publishing, 2016, pp. 1–19. DOI: 10.1007/978-3-319-46613-2_1.

[97] M. Forlingieri and T. Weilkiens. "Two Variant Modeling Methods for MBPLE at Airbus". In: *32nd Annual INCOSE International Symposium*. Detroit, MI, June 2022. DOI: 10.1002/iis2.12984.

[98] S. Lazreg et al. "Variability-Aware Design of Space Systems: Variability Modelling, Configuration Workflow and Research Directions". In: *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems*. New York, NY, USA: ACM, Feb. 2022. DOI: 10.1145/3510466.3510472.

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

[100] D.S. Raudberget, M.T. Michaelis, and H.L. Johannesson. "Combining set-based concurrent engineering and function & Means modelling to manage platform-based product family design". In: *2014 IEEE International Conference on Industrial Engineering and Engineering Management*. IEEE, Dec. 2014. DOI: 10.1109/ieem.2014.7058668.

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

[102] D.F. Wyatt et al. "Supporting product architecture design using computational design synthesis with network structure constraints". In: *Research in Engineering Design* 23.1 (Jan. 2012), pp. 17–52. DOI: 10.1007/s00163-011-0112-y.

[103] J. Gross and S. Rudolph. "Modeling graph-based satellite design languages". In: *Aerospace Science and Technology* 49 (Feb. 2016), pp. 63–72. DOI: 10.1016/j.ast.2015.11.026.

[104] S. Masfaraud et al. "Automatized gearbox architecture design exploration by exhaustive graph generation". In: *WCCM XII 2016*. Seoul, Korea, July 2016.

[105] D. Kirov et al. "ArchEx: An Extensible Framework for the Exploration of Cyber-Physical System Architectures". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. New York, NY, USA: ACM, June 2017. DOI: 10.1145/3061639.3062204.

[106] D.R. Herber, T. Guo, and J.T. Allison. "Enumeration of Architectures With Perfect Matchings". In: *Journal of Mechanical Design* 139.5 (Apr. 2017). DOI: 10.1115/1.4036132.

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

[108] G. Paparistodimou. “Generative design of robust modular system architectures”. PhD thesis. 2020. DOI: 10.48730/T9JH-6R87.

[109] P. de Vos et al. “Design space exploration for on-board energy distribution systems: A new case study”. In: *Proceedings of the 17th International Conference on Computer and IT Applications in the Maritime Industries (COMPIT ’18)*. Pavone, Italy, 2018.

[110] J. Menu, M. Nicolai, and M. Zeller. “Designing Fail-Safe Architectures for Aircraft Electrical Power Systems”. In: *2018 AIAA/IEEE Electric Aircraft Technologies Symposium*. American Institute of Aeronautics and Astronautics, July 2018. DOI: 10.2514/6.2018-5032.

[111] J. Vanhuyse. “Closed-loop Design Methodology for Hybrid Electric Vehicle Powertrains”. PhD thesis. KU Leuven, May 2018.

[112] M. Nicolai, L. Salemio, and J. Vanhuyse. “Design Space Modeling Language for the Generation of Engineering Designs”. U.S. pat. US10540477B2. Jan. 2020.

[113] I. Chakraborty and D.N. Mavris. “Integrated Assessment of Aircraft and Novel Subsystem Architectures in Early Design”. In: *54th AIAA Aerospace Sciences Meeting*. Vol. 54. 4. Reston, Virginia: American Institute of Aeronautics and Astronautics, Jan. 2016, pp. 1268–1282. DOI: 10.2514/6.2016-0215.

[114] M. Guerster and E.F. Crawley. “Dominant Suborbital Space Tourism Architectures”. In: *Journal of Spacecraft and Rockets* 34385 (June 2019), pp. 1–13. DOI: 10.2514/1.A34385.

[115] Y. Huang et al. “Architectural design space exploration of complex engineered systems with management constraints and preferences”. In: *Journal of Engineering Design* (Apr. 2024), pp. 1–32. DOI: 10.1080/09544828.2024.2335137.

[116] T. Kurtoglu and M.I. Campbell. “Automated synthesis of electromechanical design configurations from empirical analysis of function to form mapping”. In: *Journal of Engineering Design* 20.1 (Feb. 2009), pp. 83–104. DOI: 10.1080/09544820701546165.

[117] N. Albarello, J.B. Welcomme, and C. Reyterou. “A formal design synthesis and optimization method for systems architectures”. In: *Proceedings of MOSIM*. Bordeaux, France, June 2012.

[118] N.R. Shougarian. “Towards concept generation and performance-complexity tradespace exploration of engineering systems using convex hulls”. PhD thesis. Massachusetts Institute of Technology, 2017.

[119] W.L. Simmons. “A Framework for Decision Support in Systems Architecting”. PhD thesis. Massachusetts Institute of Technology, 2008, p. 198.

[120] L.K. Franzén et al. “Ontology-Represented Design Space Processing”. In: *AIAA AVIATION 2021 FORUM*. American Institute of Aeronautics and Astronautics, July 2021. DOI: 10.2514/6.2021-2426.

[121] K. Kernstine et al. “Designing for a Green Future: A Unified Aircraft Design Methodology”. In: *Journal of Aircraft* 47.5 (Sept. 2010), pp. 1789–1797. DOI: 10.2514/1.c000239.

[122] G. Apaza and D. Selva. “Automatic Composition of Encoding Scheme and Search Operators in System Architecture Optimization”. In: *41st Computers and Information in Engineering Conference (CIE)*. Virtual: American Society of Mechanical Engineers, Aug. 2021. DOI: 10.1115/detc2021-71399.

[123] P. Limbourg and H. Kochs. “Multi-objective optimization of generalized reliability design problems using feature models—A concept for early design stages”. In: *Reliability Engineering & System Safety* 93.6 (June 2008), pp. 815–828. DOI: 10.1016/j.ress.2007.03.032.

[124] R.E. Lopez-Herrejon, L. Linsbauer, and A. Egyed. “A systematic mapping study of search-based software engineering for software product lines”. In: *Information and Software Technology* 61 (May 2015), pp. 33–51. DOI: 10.1016/j.infsof.2015.01.008.

[125] S. Lazreg et al. “Multifaceted Automated Analyses for Variability-Intensive Embedded Systems”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, May 2019. DOI: 10.1109/icse.2019.00092.

[126] J. Ölvander, B. Lundén, and H. Gavel. “A computerized optimization framework for the morphological matrix applied to aircraft conceptual design”. In: *Computer-Aided Design* 41.3 (Mar. 2009), pp. 187–196. DOI: 10.1016/j.cad.2008.06.005.

[127] C.P. Frank. “A Design Space Exploration Methodology to Support Decisions under Evolving Requirements Uncertainty and its Application to Suborbital Vehicles”. PhD thesis. Georgia Institute of Technology, 2016. DOI: 10.2514/6.2015-1010.

[128] D. Selva, B. Cameron, and E.F. Crawley. “Patterns in System Architecture Decisions”. In: *Systems Engineering* 19.6 (Nov. 2016), pp. 477–497. DOI: 10.1002/sys.21370.

[129] D. Selva, B. Cameron, and E.F. Crawley. “A rule-based method for scalable and traceable evaluation of system architectures”. In: *Research in Engineering Design* 25.4 (Oct. 2014), pp. 325–349. DOI: 10.1007/s00163-014-0180-x.

[130] C.P. Frank et al. “An Evolutionary Multi-Architecture Multi-Objective Optimization Algorithm for Design Space Exploration”. In: *57th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*. January. Reston, Virginia, Jan. 2016, pp. 1–19. DOI: 10.2514/6.2016-0414.

[131] P. Saves et al. “Constrained Bayesian optimization over mixed categorical variables, with application to aircraft design”. In: *AeroBest 2021*. 2021.

[132] J. Pelamatti et al. “Bayesian optimization of variable-size design space problems”. In: *Optimization and Engineering* (July 2020). DOI: 10.1007/s11081-020-09520-z.

[133] M. Zaefferer and D. Horn. “A First Analysis of Kernels for Kriging-Based Optimization in Hierarchical Search Spaces”. In: *Parallel Problem Solving from Nature, PPSN XI*. Vol. 1. Berlin, Heidelberg: Springer Berlin Heidelberg, 2018, pp. 399–410. DOI: 10.1007/978-3-319-99259-4_32.

[134] F. Hutter and M.A. Osborne. “A Kernel for Hierarchical Parameter Spaces”. In: *arXiv* (Oct. 2013). DOI: 10.48550/ARXIV.1310.5738.

[135] M. Feurer and F. Hutter. “Hyperparameter Optimization”. In: *Automated Machine Learning*. Switzerland: Springer International Publishing, 2019, pp. 3–33. DOI: 10.1007/978-3-030-05318-5_1.

[136] J. Bergstra et al. “Algorithms for Hyper-Parameter Optimization”. In: *Advances in Neural Information Processing Systems 24*. Granada, Spain, 2011.

[137] R. Jenatton et al. “Bayesian Optimization with Tree-structured Dependencies”. In: *Proceedings of the 34th International Conference on Machine Learning*. Sydney, Australia, June 2017.

[138] O. Abdelkhalik. “Hidden Genes Genetic Optimization for Variable-Size Design Space Problems”. In: *Journal of Optimization Theory and Applications* 156.2 (Aug. 2012), pp. 450–468. DOI: 10.1007/s10957-012-0122-6.

[139] E. Talbi. “Metaheuristics for (Variable-Size) Mixed Optimization Problems: A Unified Taxonomy and Survey”. In: (Jan. 2024). DOI: 10.48550/ARXIV.2401.03880.

[140] J. Levesque et al. “Bayesian optimization for conditional hyperparameter spaces”. In: *2017 International Joint Conference on Neural Networks (IJCNN)*. Anchorage, Alaska, USA: IEEE, May 2017. DOI: 10.1109/ijcnn.2017.7965867.

[141] S. Le Digabel and S.M. Wild. “A taxonomy of constraints in black-box simulation-based optimization”. In: *Optimization and Engineering* (Sept. 2023). DOI: 10.1007/s11081-023-09839-3.

[142] D.R. Jones, M. Schonlau, and W.J. Welch. “Efficient Global Optimization of Expensive Black-Box Functions”. In: *Journal of Global Optimization* 13 (1998), pp. 455–492. DOI: 10.1023/A:1008306431147.

[143] K. Miettinen. *Nonlinear Multiobjective Optimization*. USA: Springer US, Sept. 30, 1998. 324 pp. DOI: 10.1007/978-1-4615-5563-6.

[144] O. Rudenko and M. Schoenauer. “A Steady Performance Stopping Criterion for Pareto-based Evolutionary Algorithms”. In: *6th International Multi-Objective Programming and Goal Programming Conference*. Hammamet, Tunisia, 2004.

[145] R. Priem, N. Bartoli, and Y. Diouane. “On the Use of Upper Trust Bounds in Constrained Bayesian Optimization Infill Criteria”. In: *AIAA Aviation 2019 Forum*. June. Reston, Virginia: American Institute of Aeronautics and Astronautics, June 2019, pp. 1–10. DOI: 10.2514/6.2019-2986.

[146] A.I.J. Forrester, A. Sóbester, and A.J. Keane. “Optimization with missing data”. In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 462.2067 (Jan. 2006), pp. 935–945. DOI: 10.1098/rspa.2005.1608.

[147] J. Müller and M. Day. “Surrogate Optimization of Computationally Expensive Black-Box Problems with Hidden Constraints”. In: *INFORMS Journal on Computing* 31.4 (Oct. 2019), pp. 689–702. DOI: 10.1287/ijoc.2018.0864.

[148] M.D. Krengel and M. Hepperle. “Effects of Wing Elasticity and Basic Load Alleviation on Conceptual Aircraft Designs”. In: *AIAA SCITECH 2022 Forum*. Jan. 2022. DOI: 10.2514/6.2022-0126.

[149] L. Yang and A. Shami. “On hyperparameter optimization of machine learning algorithms: Theory and practice”. In: *Neurocomputing* 415 (Nov. 2020), pp. 295–316. DOI: 10.1016/j.neucom.2020.07.061.

[150] D.R. Jones and J.R.R.A. Martins. “The DIRECT algorithm: 25 years Later”. In: *Journal of Global Optimization* 79.3 (Oct. 2020), pp. 521–566. DOI: 10.1007/s10898-020-00952-6.

[151] M. Locatelli and F. Schoen. “(Global) Optimization: Historical notes and recent developments”. In: *EURO Journal on Computational Optimization* 9 (2021), p. 100012. DOI: 10.1016/j.ejco.2021.100012.

[152] F. Glover and G. Kochenberger. *Handbook of Metaheuristics*. Vol. 57. 2003, pp. 457–474–474. DOI: 10.1007/b101874.

[153] A. Petrowski and S. Ben-Hamida. *Evolutionary Algorithms. An Overview*. London, UK: John Wiley & Sons, 2017, p. 256. ISBN: 978-184821804-8.

[154] R. Hamano et al. “CMA-ES with margin”. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. Boston, US: ACM, July 2022. DOI: 10.1145/3512290.3528827.

[155] K. Deb et al. “A fast and elitist multiobjective genetic algorithm: NSGA-II”. In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197. DOI: 10.1109/4235.996017.

[156] H.M. Nyew, O. Abdelkhalik, and N. Onder. “Structured-Chromosome Evolutionary Algorithms for Variable-Size Autonomous Interplanetary Trajectory Planning Optimization”. In: *Journal of Aerospace Information Systems* 12.3 (Mar. 2015), pp. 314–328. DOI: 10.2514/1.i010272.

[157] R.S. Zebulum, M. Vellasco, and M.A. Pacheco. “Variable Length Representation in Evolutionary Electronics”. In: *Evolutionary Computation* 8.1 (Mar. 2000), pp. 93–120. DOI: 10.1162/106365600568112.

[158] N. Hitomi and D. Selva. “Constellation optimization using an evolutionary algorithm with a variable-length chromosome”. In: *2018 IEEE Aerospace Conference*. IEEE, Mar. 2018. DOI: 10.1109/aero.2018.8396743.

[159] J. Gamot et al. “Hidden-variables genetic algorithm for variable-size design space optimal layout problems with application to aerospace vehicles”. In: *Engineering Applications of Artificial Intelligence* 121 (May 2023), p. 105941. DOI: 10.1016/j.engappai.2023.105941.

[160] S. Gratton and L.N. Vicente. “A Merit Function Approach for Direct Search”. In: *SIAM Journal on Optimization* 24.4 (Jan. 2014), pp. 1980–1998. DOI: 10.1137/130917661.

[161] A. Aleti et al. “Software Architecture Optimization Methods: A Systematic Literature Review”. In: *IEEE Transactions on Software Engineering* 39.5 (May 2013), pp. 658–683. DOI: 10.1109/tse.2012.64.

[162] M.A. Buonanno. “A method for aircraft concept exploration using multicriteria interactive genetic algorithms”. PhD thesis. Georgia Institute of Technology, Aug. 2005.

[163] D.J. Pate, M.D. Patterson, and B.J. German. “Optimizing Families of Reconfigurable Aircraft for Multiple Missions”. In: *Journal of Aircraft* 49.6 (Nov. 2012), pp. 1988–2000. DOI: 10.2514/1.C031667.

[164] C.P. Frank et al. “Evolutionary multi-objective multi-architecture design space exploration methodology”. In: *Optimization and Engineering* 19.2 (Jan. 2018), pp. 359–381. DOI: 10.1007/s11081-018-9373-x.

[165] T. Chugh et al. “A survey on handling computationally expensive multiobjective optimization problems with evolutionary algorithms”. In: *Soft Computing* 23.9 (May 2019), pp. 3137–3166. DOI: 10.1007/s00500-017-2965-0.

[166] A.I.J. Forrester, A. Sóbester, and A.J. Keane. *Engineering Design via Surrogate Modelling*. 2008. DOI: 10.1002/9780470770801.

[167] R. Garnett. *Bayesian Optimization*. Cambridge, UK: Cambridge University Press, Jan. 2023. DOI: 10.1017/9781108348973.

[168] C. Rasmussen and C. Williams. *Gaussian Processes for Machine Learning*. Vol. 14. 2. the MIT Press, 2006, pp. 69–106. ISBN: 026218253X.

[169] D.G. Krige. “A statistical approach to some basic mine valuation problems on the Witwatersrand”. In: *Journal of the Southern African Institute of Mining and Metallurgy* 52.6 (1951), pp. 119–139. DOI: 10.10520/AJA0038223X_4792.

[170] M. Schonlau, W.J. Welch, and D.R. Jones. “Global versus local search in constrained optimization of computer models”. In: *Lecture Notes - Monograph Series*. online: Institute of Mathematical Statistics, 1998, pp. 11–25. DOI: 10.1214/lnms/1215456182.

[171] M.J. Sasena. “Flexibility and Efficiency Enhancements for Constrained Global Design Optimization with Kriging Approximations”. PhD thesis. University of Michigan, 2002, pp. 1–237.

[172] J. Knowles. “ParEGO: a hybrid algorithm with on-line landscape approximation for expensive multiobjective optimization problems”. In: *IEEE Transactions on Evolutionary Computation* 10.1 (Feb. 2006), pp. 50–66. DOI: 10.1109/TEVC.2005.851274.

[173] S. Rojas-Gonzalez and I. Van Nieuwenhuyse. “A Survey on Kriging-Based Infill Algorithms for Multiobjective Simulation Optimization”. In: *Computers & Operations Research* 116 (Apr. 2019), p. 104869. DOI: 10.1016/j.cor.2019.104869.

[174] P. Saves et al. “A mixed-categorical correlation kernel for Gaussian process”. In: *Neurocomputing* 550 (Sept. 2023), p. 126472. DOI: 10.1016/j.neucom.2023.126472.

[175] E.C. Garrido-Merchán and D. Hernández-Lobato. “Dealing with categorical and integer-valued variables in Bayesian Optimization with Gaussian processes”. In: *Neurocomputing* 380 (Mar. 2020), pp. 20–35. DOI: 10.1016/j.neucom.2019.11.004.

[176] S. Daulton et al. “Bayesian Optimization over Discrete and Mixed Spaces via Probabilistic Reparameterization”. In: (Oct. 2022). DOI: 10.48550/ARXIV.2210.10199.

[177] J. Pelamatti et al. “Efficient global optimization of constrained mixed variable problems”. In: *Journal of Global Optimization* 73.3 (Mar. 2019), pp. 583–613. DOI: 10.1007/s10898-018-0715-1.

[178] M.M. Zuniga and D. Sinoquet. “Global optimization for mixed categorical-continuous variables based on Gaussian process models with a randomized categorical space exploration step”. In: *INFOR: Information Systems and Operational Research* 58.2 (Mar. 2020), pp. 310–341. DOI: 10.1080/03155986.2020.1730677.

[179] K. Dreczkowski, A. Grosnit, and H.B. Ammar. “Framework and Benchmarks for Combinatorial and Mixed-variable Bayesian Optimization”. In: (June 2023). DOI: 10.48550/ARXIV.2306.09803.

[180] A. Yousefpour et al. “GP+: A Python library for kernel-based learning via Gaussian processes”. In: *Advances in Engineering Software* 195 (Sept. 2024), p. 103686. DOI: 10.1016/j.advengsoft.2024.103686.

[181] C. Audet, E. Hallé-Hannan, and S. Le Digabel. “A general mathematical framework for constrained mixed-variable blackbox optimization problems with meta and categorical variables”. In: (Apr. 2022). DOI: 10.48550/ARXIV.2204.00886.

[182] X. Lu et al. “Structured Variationally Auto-encoded Optimization”. In: *Proceedings of the 35th International Conference on Machine Learning*. Stockholm, SE: PMLR, Oct. 2018.

[183] D. Horn et al. “Surrogates for hierarchical search spaces”. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. Prague, CZ: ACM, July 2019. DOI: 10.1145/3321707.3321765.

[184] M.A. Bouhlel et al. “Efficient global optimization for high-dimensional constrained problems by using the Kriging models combined with the partial least squares method”. In: *Engineering Optimization* 50.12 (Feb. 2018), pp. 2038–2053. DOI: 10.1080/0305215x.2017.1419344.

[185] R. Priem et al. “High-dimensional efficient global optimization using both random and supervised embeddings”. In: *AIAA AVIATION 2023 Forum*. San Diego, CA, USA: American Institute of Aeronautics and Astronautics, June 2023. DOI: 10.2514/6.2023-4448.

[186] P. Saves et al. “High-dimensional mixed-categorical Gaussian processes with application to multidisciplinary design optimization for a green aircraft”. In: *Structural and Multidisciplinary Optimization* 67.5 (May 2024). DOI: 10.1007/s00158-024-03785-z.

[187] S. Valencia Ibañez. “Optimization Strategies for System Architecting Problems”. MSc thesis. Delft University of Technology, Aug. 2023.

[188] E. Hallé-Hannan et al. “A graph-structured distance for heterogeneous datasets with meta variables”. In: *Optimization Online* (May 2024).

[189] Object Management Group. *SysPhS: SysML Extension for Physical Interaction and Signal Flow Simulation*. 2021. URL: https://www.omg.org/spec/SysPhS.

[190] C.J.J. Paredis et al. “An overview of the SysML-Modelica transformation specification”. In: *INCOSE International Symposium* 2 (2010), pp. 1–14. DOI: 10.1002/j.2334-5837.2010.tb01099.x.

[191] M. Guenov et al. “Aircraft Systems Architecting - Logical-Computational Domains Interface”. In: *31st Congress of the International Council of the Aeronautical Sciences* (2018), pp. 1–12.

[192] A. Raju Kulkarni et al. “An MBSE approach to support Knowledge Based Engineering application development”. Eng. In: (2023). DOI: 10.13009/EUCASS2023-495.

[193] B.I. Min, A.A. Kerzhner, and C.J.J. Paredis. “Process Integration and Design Optimization for Model-Based Systems Engineering With SysML”. In: *Volume 2: 31st Computers and Information in Engineering Conference, Parts A and B*. Washington, DC, USA: ASME, 2011, pp. 1361–1369. DOI: 10.1115/DETC2011-48453.

[194] Y. Bile et al. “Towards Automating the Sizing Process in Conceptual (Airframe) Systems Architecting”. In: *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference* January (2018), pp. 1–24. DOI: 10.2514/6.2018-1067.

[195] C. Gomes et al. “Co-Simulation: A Survey”. In: *ACM Computing Surveys* 51.3 (May 2018), pp. 1–33. DOI: 10.1145/3179993.

[196] A. Junghanns et al. “The Functional Mock-up Interface 3.0 - New Features Enabling New Applications”. In: *Proceedings of 14th Modelica Conference 2021, Linköping, Sweden, September 20-24, 2021*. Modelica 2021. Linköping University Electronic Press, Sept. 2021. DOI: 10.3384/ecp2118117.

[197] R. Hällqvist et al. “Heterogeneous Systems Modelling in Support of Incremental Development”. In: *33rd Congress of the International Council of the Aeronautical Sciences, ICAS 2022*. Stockholm, SE, Sept. 2022.

[198] S. Balestrini-Robinson, D.F. Freeman, and D.C. Browne. “An Object-oriented and Executable SysML Framework for Rapid Model Development”. In: *Procedia Computer Science* 44 (2015), pp. 423–432. DOI: 10.1016/j.procs.2015.03.062.

[199] T.F. Beernaert and L.F.P. Etman. “Multi-level Decomposed Systems Design: Converting a Requirement Specification into an Optimization Problem”. In: *Proceedings of the Design Society: International Conference on Engineering Design* 1.1 (July 2019), pp. 3691–3700. DOI: 10.1017/dsi.2019.376.

[200] J. Au and R. Ravindranath. “Bridging the Gap Between Architects, Engineers and Other Stakeholders in Complex and Multidisciplinary Systems – a Holistic, Inclusive and Interactive Design Approach”. In: *INCOSE International Symposium* 30.1 (July 2020), pp. 153–167. DOI: 10.1002/j.2334-5837.2020.00714.x.

[201] A.K. Jeyaraj, N. Tabesh, and S. Liscouet-Hanke. “Connecting Model-based Systems Engineering and Multidisciplinary Design Analysis and Optimization for Aircraft Systems Architecting”. In: *AIAA AVIATION 2021 FORUM*. Virtual Event: American Institute of Aeronautics and Astronautics, Aug. 2021. DOI: 10.2514/6.2021-3077.

[202] J. Ross et al. “Applying Model-Based Systems Engineering Methods to a Novel Shared Systems Simulation Methodology”. In: *32nd Annual INCOSE International Symposium*. Detoit, MI, June 2022. DOI: 10.1002/iis2.12996.

[203] R. Demachy et al. “A Model-Based System Engineering Approach for Multi-Disciplinary Analysis and Design Optimisation Processes Definition”. In: *The Complex Systems Design & Management Conference (CSD&M 2022)*. 2022.

[204] R. Swaminathan, D. Sarojini, and J.T. Hwang. “Integrating MBSE and MDO through an Extended Requirements-Functional-Logical-Physical (RFLP) Framework”. In: *AIAA AVIATION 2023 Forum*. San Diego, CA: American Institute of Aeronautics and Astronautics, June 2023. DOI: 10.2514/6.2023-3908.

[205] A. Busch, E. Vidana, and A. Luc. “Connecting MBSE to Spacecraft Modeling & Simulation”. In: *MBSE2024 Workshop*. Bremen, Germany, May 2024.

[206] P.D. Ciampa and B. Nagel. “Towards the 3rd Generation MDO Collaborative Environment”. In: *30th Congress of the International Council of the Aeronautical Sciences*. Daejeon, Korea, 2016, pp. 1–12.

[207] M. Alder et al. “Recent Advances in Establishing a Common Language for Aircraft Design with CPACS”. In: *Aerospace Europe Conference*. 2020.

[208] I. van Gent. “Agile MDAO Systems: A Graph-based Methodology to Enhance Collaborative Multidisciplinary Design”. PhD thesis. Delft University of Technology, 2019.

[209] I. van Gent and G. La Rocca. “Formulation and integration of MDAO systems for collaborative design: A graph-based methodological approach”. In: *Aerospace Science and Technology* 90 (July 2019), pp. 410–433. DOI: 10.1016/j.ast.2019.04.039.

[210] J.R.R.A. Martins and A.B. Lambe. “Multidisciplinary Design Optimization: A Survey of Architectures”. In: *AIAA Journal* 51.9 (2013), pp. 2049–2075. DOI: 10.2514/1.J051895.

[211] I. van Gent et al. “Knowledge architecture supporting collaborative MDO in the AGILE paradigm”. In: *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference* June (2017), pp. 5–9. DOI: 10.2514/6.2017-4139.

[212] P. Helle et al. “Enabling Multidisciplinary-Analysis of SysML Models in a Heterogeneous Tool Landscape using Parametric Analysis Models”. In: *The Complex Systems Design & Management Conference (CSD&M 2022)*. 2022.

[213] A. Bruggeman et al. “Model-Based Approach for the Simultaneous Design of Airframe Components and their Production Process Using Dynamic MDAO Workflows”. In: *AIAA SCITECH 2024 Forum*. American Institute of Aeronautics and Astronautics, Jan. 2024. DOI: 10.2514/6.2024-1530.

[214] J.S. Sonneveld et al. “Dynamic workflow generation applied to aircraft moveable architecture optimization”. Eng. In: (2023). DOI: 10.13009/EUCASS2023-544.

[215] Stewart Greenhill et al. “Bayesian Optimization for Adaptive Experimental Design: A Review”. In: *IEEE Access* 8 (2020), pp. 13937–13948. DOI: 10.1109/access.2020.2966228.

[216] J. Görtler, R. Kehlbeck, and O. Deussen. “A Visual Exploration of Gaussian Processes”. In: *Distill* 4.4 (Apr. 2019). DOI: 10.23915/distill.00017.

[217] A.I. Cowen-Rivers et al. “HEBO: Pushing The Limits of Sample-Efficient Hyperparameter Optimisation”. In: (Dec. 2020). DOI: 10.48550/ARXIV.2012.03826.

[218] W. Lyu et al. “Batch Bayesian Optimization via Multi-objective Acquisition Ensemble for Automated Analog Circuit Design”. In: *Proceedings of the 35th International Conference on Machine Learning*. Stockholm, SE: PMLR, Oct. 2018.

[219] D.D. Cox and S. John. “A statistical method for global optimization”. In: *1992 IEEE International Conference on Systems, Man, and Cybernetics*. Chicago, IL, USA: IEEE, 1992. DOI: 10.1109/icsmc.1992.271617.

[220] G.I. Hawe and J.K. Sykulski. “An Enhanced Probability of Improvement Utility Function for Locating Pareto Optimal Solutions”. In: *16th Conference on the Computation of Electromagnetic Fields, COMPUMAG, Aachen, Germany* 3 (2007), pp. 965–966.

[221] A.A.M. Rahat, R.M. Everson, and J.E. Fieldsend. “Alternative infill strategies for expensive multi-objective optimisation”. In: *Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO ’17*. New York, USA: ACM Press, 2017, pp. 873–880. DOI: 10.1145/3071178.3071276.

[222] M. Sohst, F. Afonso, and A. Suleman. “Surrogate-based optimization based on the probability of feasibility”. In: *Structural and Multidisciplinary Optimization* 65.1 (Dec. 2021). DOI: 10.1007/s00158-021-03134-4.

[223] D. Benavides, S. Segura, and A. Ruiz-Cortés. “Automated analysis of feature models 20 years later: A literature review”. In: *Information Systems* 35.6 (Sept. 2010), pp. 615–636. DOI: 10.1016/j.is.2010.01.001.

[224] J.S. Gray. “MDO Problem Suite v3: We have the technology, we can rebuilt it!” In: *AIAA AVIATION 2020 FORUM*. Virtual Event, June 2020.

[225] M. Renardy et al. “To Sobol or not to Sobol? The effects of sampling schemes in systems biology applications”. In: *Mathematical Biosciences* 337 (July 2021), p. 108593. DOI: 10.1016/j.mbs.2021.108593.

[226] C. Kaltenecker et al. “Distance-Based Sampling of Software Configuration Spaces”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, CA: IEEE, May 2019. DOI: 10.1109/icse.2019.00112.

[227] Y. Collette et al. “Multidisciplinary Design Optimization in Computational Mechanics”. In: ed. by P. Breitkopf and R. Filomeno Coelho. London, UK: Wiley, Feb. 2013. Chap. Object-Oriented Programming of Optimizers – Examples in Scilab, pp. 499–538. DOI: 10.1002/9781118600153.ch14.

[228] S. Salcedo-Sanz. “A survey of repair methods used as constraint handling techniques in evolutionary algorithms”. In: *Computer Science Review* 3.3 (Aug. 2009), pp. 175–192. DOI: 10.1016/j.cosrev.2009.07.001.

[229] M. Balandat et al. “BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization”. In: *Advances in Neural Information Processing Systems 33* (Oct. 2019). DOI: 10.48550/ARXIV.1910.06403.

[230] M. Lindauer et al. “BOAH: A Tool Suite for Multi-Fidelity Bayesian Optimization & Analysis of Hyperparameters”. In: (Aug. 2019). DOI: 10.48550/ARXIV.1908.06756.

[231] W. Huyer and A. Neumaier. “SNOBFIT – Stable Noisy Optimization by Branch and Fit”. In: *ACM Transactions on Mathematical Software* 35.2 (July 2008), pp. 1–25. DOI: 10.1145/1377612.1377613.

[232] H.K.H. Lee et al. *Optimization Subject to Hidden Constraints via Statistical Emulation*. Tech. rep. UCSC-SOE-10-10. UC Santa Cruz, Apr. 2010.

[233] A. Marchildon and D. Zingg. “Gradient-Enhanced Bayesian Optimization With Application to Aerodynamic Shape Optimization”. In: *AIAA AVIATION FORUM AND ASCEND 2024*. Las Vegas, NV, USA, July 2024. DOI: 10.2514/6.2024-4405.

[234] A.P. Roberts et al. “Multi-objective Bayesian shape optimization of an industrial hydrodynamic separator using unsteady Eulerian-Lagrangian simulations”. In: *Optimization and Engineering* (Aug. 2024). DOI: 10.1007/s11081-024-09907-2.

[235] S.R. Alimo, P. Beyhaghi, and T.R. Bewley. “Delaunay-Based Global Optimization in Nonconvex Domains Defined by Hidden Constraints”. In: *Computational Methods in Applied Sciences*. Springer International Publishing, Sept. 2018, pp. 261–271. DOI: 10.1007/978-3-319-89890-2_17.

[236] M. Sacher et al. “A classification approach to efficient global optimization in presence of non-computable domains”. In: *Structural and Multidisciplinary Optimization* 58.4 (Apr. 2018), pp. 1537–1557. DOI: 10.1007/s00158-018-1981-8.

[237] M.A. Gelbart, J. Snoek, and R.P. Adams. “Bayesian Optimization with Unknown Constraints”. In: *UAI’14: Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence*. Arlington, VA, USA, Mar. 22, 2014.

[238] F. Bachoc, C. Helbert, and V. Picheny. “Gaussian process optimization with failures: classification and convergence proof”. In: *Journal of Global Optimization* 78.3 (July 2020), pp. 483–506. DOI: 10.1007/s10898-020-00920-0.

[239] A. Tfaily et al. “Bayesian optimization with hidden constraints for aircraft design”. In: *Structural and Multidisciplinary Optimization* 67.7 (July 2024). DOI: 10.1007/s00158-024-03833-8.

[240] C. Audet, G. Caporossi, and S. Jacquet. “Binary, unrelaxable and hidden constraints in blackbox optimization”. In: *Operations Research Letters* 48.4 (July 2020), pp. 467–471. DOI: 10.1016/j.orl.2020.05.011.

[241] P. Saves et al. “A general square exponential kernel to handle mixed-categorical variables for Gaussian process”. In: *AIAA AVIATION 2022 Forum*. Chicago, IL, USA: American Institute of Aeronautics and Astronautics, June 2022. DOI: 10.2514/6.2022-3870.

[242] J. Hensman, A. Matthews, and Z. Ghahramani. “Scalable Variational Gaussian Process Classification”. In: *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*. Vol. 38. Proceedings of Machine Learning Research. San Diego, California, USA: PMLR, Sept. 2015, pp. 351–360.

[243] J. Pelamatti et al. “Overview and Comparison of Gaussian Process-Based Surrogate Models for Mixed Continuous and Discrete Variables: Application on Aerospace Design Problems”. In: *High-Performance Simulation-Based Optimization*. Vol. 833. Studies in Computational Intelligence. Cham: Springer International Publishing, June 2020, pp. 189–224. DOI: 10.1007/978-3-030-18764-4_9.

[244] J. Blank and K. Deb. “Pymoo: Multi-Objective Optimization in Python”. In: *IEEE Access* 8 (2020), pp. 89497–89509. DOI: 10.1109/access.2020.2990567.

[245] V. Picheny et al. “Trieste: Efficiently Exploring The Depths of Black-box Functions with TensorFlow”. In: (Feb. 2023). DOI: 10.48550/ARXIV.2302.08436.

[246] N. Bartoli et al. “Adaptive modeling strategy for constrained global optimization with application to aerodynamic wing design”. In: *Aerospace Science and Technology* 90 (July 2019), pp. 85–102. DOI: 10.1016/j.ast.2019.03.041.

[247] P. Bekemeyer et al. “Data-Driven Aerodynamic Modeling Using the DLR SMARTy Toolbox”. In: *AIAA AVIATION 2022 Forum*. Chicago, USA: American Institute of Aeronautics and Astronautics, June 2022. DOI: 10.2514/6.2022-3899.

[248] J.S. Gray et al. “OpenMDAO: an open-source framework for multidisciplinary design, analysis, and optimization”. In: *Structural and Multidisciplinary Optimization* 59.4 (Mar. 2019), pp. 1075–1104. DOI: 10.1007/s00158-019-02211-z.

[249] E.S. Hendricks and J.S. Gray. “pyCycle: A Tool for Efficient Optimization of Gas Turbine Engine Cycles”. In: *Aerospace* 6.8 (Aug. 2019), p. 87. DOI: 10.3390/aerospace6080087.

[250] C. Bauer et al. “Flight Control System Architecture Optimization for Fly-By-Wire Airliners”. In: *Journal of Guidance, Control, and Dynamics* 30.4 (July 2007), pp. 1023–1029. DOI: 10.2514/1.26311.

[251] E. Weinberger. “Correlated and uncorrelated fitness landscapes and how to tell the difference”. In: *Biological Cybernetics* 63.5 (Sept. 1990), pp. 325–336. DOI: 10.1007/bf00202749.

[252] J.L. Rodgers and W.A. Nicewander. “Thirteen Ways to Look at the Correlation Coefficient”. In: *The American Statistician* 42.1 (Feb. 1988), p. 59. DOI: 10.2307/2685263.

[253] P.A. Fishwick. In: *IIE Transactions* 30.9 (1998), pp. 811–820. DOI: 10.1023/a:1007548116679.

[254] B.J. Brelje and J.R.R.A. Martins. “Electric, hybrid, and turboelectric fixed-wing aircraft: A review of concepts, models, and design approaches”. In: *Progress in Aerospace Sciences* 104 (Jan. 2019), pp. 1–19. DOI: 10.1016/j.paerosci.2018.06.004.

[255] S. Biser et al. “Design Space Exploration Study and Optimization of a Distributed Turbo-Electric Propulsion System for a Regional Passenger Aircraft”. In: *AIAA Propulsion and Energy 2020 Forum*. American Institute of Aeronautics and Astronautics, Aug. 2020. DOI: 10.2514/6.2020-3592.

[256] B.J. Brelje and J.R.R.A. Martins. “Development of a Conceptual Design Model for Aircraft Electric Propulsion with Efficient Gradients”. In: *2018 AIAA/IEEE Electric Aircraft Technologies Symposium*. American Institute of Aeronautics and Astronautics, July 2018. DOI: 10.2514/6.2018-4979.

[257] S. Wöhler et al. “Preliminary Aircraft Design within a Multi-disciplinary and Multi-fidelity Design Environment”. In: *Aerospace Europe Conference, AEC2020-032*. 2020.

[258] A.B. Lambe and J.R.R.A. Martins. “Extensions to the design structure matrix for the description of multidisciplinary design, analysis, and optimization processes”. In: *Structural and Multidisciplinary Optimization* 46.2 (2012), pp. 273–284. DOI: 10.1007/s00158-012-0763-y.

[259] G. Esdras and S. Liscouët-Hanke. “Development of Core Functions for Aircraft Conceptual Design: Methodology and Results”. In: *Canadian Aeronautics and Space Institute Conference*. Montreal, CA, 2015.

[260] R.P. Grimaldi. *Discrete and Combinatorial Mathematics: An Applied Introduction*. Pearson Education, 2004.

[261] R. García Sánchez. “Adaptation of an MDO Platform for System Architecture Optimization”. MSc thesis. Delft, NL: Delft University of Technology, Jan. 2024.

[262] M. Fioriti et al. “Multidisciplinary design of a more electric regional aircraft including certification constraints”. In: *AIAA AVIATION 2022 Forum*. American Institute of Aeronautics and Astronautics, June 2022. DOI: 10.2514/6.2022-3932.

[263] B. Aigner et al. “Graph-based algorithms and data-driven documents for formulation and visualization of large MDO systems”. In: *CEAS Aeronautical Journal* 0.0 (2018), p. 0. DOI: 10.1007/s13272-018-0312-5.

[264] I. van Gent, G. La Rocca, and M.F.M. Hoogreef. “CMDOWS: a proposed new standard to store and exchange MDO systems”. In: *CEAS Aeronautical Journal* 9.4 (2018), pp. 607–627. DOI: 10.1007/s13272-018-0307-2.

[265] B. Boden et al. “RCE: An Integration Environment for Engineering and Science”. In: *SoftwareX* 15 (July 2021), p. 100759. DOI: 10.1016/j.softx.2021.100759.

[266] S.C. Brailsford, C.N. Potts, and B.M. Smith. “Constraint satisfaction problems: Algorithms and applications”. In: *European Journal of Operational Research* 119.3 (Dec. 1999), pp. 557–581. DOI: 10.1016/s0377-2217(98)00364-6.

[267] C. Barrett and C. Tinelli. “Satisfiability Modulo Theories”. In: *Handbook of Model Checking*. Springer International Publishing, 2018, pp. 305–343. DOI: 10.1007/978-3-319-10575-8_11.

[268] J. Bredin. “Top-down Functional Composition”. In: *30th Annual INCOSE International Symposium*. INCOSE. Virtual Event, July 2020. DOI: 10.1002/j.2334-5837.2020.00759.x.

[269] L. Knöös Franzén. *A System of Systems View in Early Product Development : An Ontology-Based Approach*. Linköping University Electronic Press, June 2023. DOI: 10.3384/9789180751667.

[270] O.C. Eichmann et al. “Development of functional architectures for cyber-physical systems using interconnectable models”. In: *Systems Engineering* (May 2024). DOI: 10.1002/sys.21761.

[271] M.F. Austin and D.M. York. “System Readiness Assessment (SRA) an Illustrative Example”. In: *Procedia Computer Science* 44 (2015), pp. 486–496. DOI: 10.1016/j.procs.2015.03.031.

[272] T. Garg et al. “Using TRLs and System Architecture to Estimate Technology Integration Risk”. In: *21st International Conference on Engineering Design, ICED17*. August. Vancouver, CA, 2017.

[273] K. Sinha and O.L. de Weck. “A network-based structural complexity metric for engineered complex systems”. In: *2013 IEEE International Systems Conference (SysCon)*. IEEE, Apr. 2013, pp. 426–430. DOI: 10.1109/SysCon.2013.6549917.

[274] J.D. Summers and J.J. Shah. “Mechanical Engineering Design Complexity Metrics: Size, Coupling, and Solvability”. In: *Journal of Mechanical Design* 132.2 (2010), p. 021004. DOI: 10.1115/1.4000759.

[275] IEEE/ISO/IEC. *International Standard 42010-2022 - Software, systems and enterprise – Architecture description*. IEEE, 2022. DOI: 10.1109/ieeestd.2022.9938446.

[276] L. Regenwetter, A.H. Nobari, and G. Ahmed. “Deep Generative Models in Engineering Design: A Review”. In: *Journal of Mechanical Design* 144.7 (Mar. 2022). DOI: 10.1115/1.4053859.

[277] S. Feuerriegel et al. “Generative AI”. In: *Business & Information Systems Engineering* 66.1 (Sept. 2023), pp. 111–126. DOI: 10.1007/s12599-023-00834-7.

[278] G. Apaza and D. Selva. “Leveraging Large Language Models for Tradespace Exploration”. In: *Journal of Spacecraft and Rockets* (May 2024), pp. 1–19. DOI: 10.2514/1.a35834.

[279] A. Jeyaraj and S. Liscouet-Hanke. “A model-based systems engineering approach for efficient flight control system architecture variants modelling in conceptual design”. In: *Recent Advances in Aerospace Actuation Systems and Components*. Toulouse, France, May 2018.

[280] B. Annighöfer and F. Thielecke. “A systems architecting framework for optimal distributed integrated modular avionics architectures”. In: *CEAS Aeronautical Journal* 6.3 (May 2015), pp. 485–496. DOI: 10.1007/s13272-015-0156-1.

[281] L. Knöös Franzén et al. “An Ontological Approach to System of Systems Engineering in Product Development”. In: *Proceedings of the 10th Aerospace Technology Congress, October 8-9, 2019, Stockholm, Sweden*. Linköping University Electronic Press, Oct. 2019. DOI: 10.3384/ecp19162004.

[282] J. Page Risueño and B. Nagel. “Development of a Knowledge-Based Engineering Framework for Modeling Aircraft Production”. In: *AIAA Aviation 2019 Forum*. June. Reston, Virginia: American Institute of Aeronautics and Astronautics, June 2019. DOI: 10.2514/6.2019-2889.

[283] C. Ying et al. “NAS-Bench-101: Towards Reproducible Neural Architecture Search”. In: *Proceedings of the 36th International Conference on Machine Learning*. Long Beach, CA, USA: PMLR, Sept. 2019.

[284] Q. Zhang et al. “Retiarii: A Deep Learning Exploratory-Training Framework”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. Virtual: USENIX Association, Nov. 2020, pp. 919–936. ISBN: 978-1-939133-19-9.

[285] M. Fowler. *Domain-specific languages*. Upper Saddle River, N.J.: Addison-Wesley, 2011. 1597 pp. ISBN: 0132107546.

[286] G. Donelli et al. “A Value-driven Concurrent Approach for Aircraft Design-Manufacturing-Supply Chain”. In: *Production & Manufacturing Research* 11.1 (Nov. 2023). DOI: 10.1080/21693277.2023.2279709.

[287] M.A. Bouhlel et al. “Improving Kriging surrogates of high-dimensional design models by Partial Least Squares dimension reduction”. In: *Structural and Multidisciplinary Optimization* 53.5 (May 2016), pp. 935–952. DOI: 10.1007/s00158-015-1395-9.

[288] R. Calandra et al. “Manifold Gaussian Processes for regression”. In: *2016 International Joint Conference on Neural Networks (IJCNN)*. Vancouver, CA: IEEE, July 2016. DOI: 10.1109/ijcnn.2016.7727626.

[289] Z. Xu and S. Zhe. “Standard Gaussian Process is All You Need for High-Dimensional Bayesian Optimization”. In: (Feb. 2024). DOI: 10.48550/ARXIV.2402.02746.

[290] M. Binois et al. “The Kalai-Smorodinsky solution for many-objective Bayesian optimization”. In: *Journal of Machine Learning Research* 21.150 (2020), pp. 1–42.

[291] L.A. Martín and E.C. Garrido-Merchán. “Many Objective Bayesian Optimization”. In: (July 2021). DOI: 10.48550/ARXIV.2107.04126.

[292] R. Charayron et al. “Multi-fidelity Bayesian optimization strategy applied to Overall Drone Design”. In: *AIAA SCITECH 2023 Forum*. National Harbor, MD, USA: American Institute of Aeronautics and Astronautics, Jan. 2023. DOI: 10.2514/6.2023-2366.

[293] IEEE/ISO/IEC. *International Standard 42030-2019 - Software, systems and enterprise – Architecture evaluation framework*. IEEE, 2019. DOI: 10.1109/ieeestd.2019.8767001.

[294] L. Halsema. “An Automated Vehicle System Architecture Exploration, Evaluation and Uncertainty-Based Design Optimization Framework”. MSc thesis. Delft University of Technology, Aug. 2024.

[295] P. Yanardag and S.V.N. Vishwanathan. “Deep Graph Kernels”. In: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’15. ACM, Aug. 2015. DOI: 10.1145/2783258.2783417.

[296] N.M. Kriege, F.D. Johansson, and C. Morris. “A survey on graph kernels”. In: *Applied Network Science* 5.1 (Jan. 2020). DOI: 10.1007/s41109-019-0195-3.

[297] G. Nikolentzos, G. Siglidis, and M. Vazirgiannis. “Graph Kernels: A Survey”. In: *Journal of Artificial Intelligence Research* 72 (Nov. 2021), pp. 943–1027. DOI: 10.1613/jair.1.13225.

[298] A. Sirico and D.R. Herber. “On the Use of Geometric Deep Learning for the Iterative Classification and Down-Selection of Analog Electric Circuits”. In: *Journal of Mechanical Design* 146.5 (Nov. 2023). DOI: 10.1115/1.4063659.

[299] R. Griffiths et al. “GAUCHE: A Library for Gaussian Processes in Chemistry”. In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh et al. Vol. 36. Curran Associates, Inc., 2023, pp. 76923–76946.

[300] M. Fey and J.E. Lenssen. “Fast Graph Representation Learning with PyTorch Geometric”. In: (Mar. 2019). DOI: 10.48550/ARXIV.1903.02428.

[301] S. Ok. “A graph similarity for deep learning”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 1–12.

[302] Z. Wu et al. “A Comprehensive Survey on Graph Neural Networks”. In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1 (Jan. 2021), pp. 4–24. DOI: 10.1109/tnnls.2020.2978386.

[303] S. Dubreuil et al. “Efficient global multidisciplinary optimization based on surrogate models”. In: *2018 Multidisciplinary Analysis and Optimization Conference* (2018). DOI: 10.2514/6.2018-3745.

[304] I. Cardoso et al. “Constrained efficient global multidisciplinary design optimization using adaptive disciplinary surrogate enrichment”. In: *Structural and Multidisciplinary Optimization* 67.2 (Feb. 2024). DOI: 10.1007/s00158-023-03736-0.

[305] F. Gallard et al. “GEMS, a Generic Engine for MDO Scenarios: Key Features in Application”. In: *AIAA Aviation 2019 Forum*. June. Reston, Virginia: American Institute of Aeronautics and Astronautics, June 2019, pp. 1–15. DOI: 10.2514/6.2019-2991.

[306] E. Lac et al. “CoSApp: a Python library to create, simulate and design complex systems.” In: *Journal of Open Source Software* 9.94 (Feb. 2024), p. 6292. DOI: 10.21105/joss.06292.

[307] V. Gandarillas et al. “A graph-based methodology for constructing computational models that automates adjoint-based sensitivity analysis”. In: *Structural and Multidisciplinary Optimization* 67.5 (May 2024). DOI: 10.1007/s00158-024-03792-0.

[308] P.D. Ciampa, G. La Rocca, and B. Nagel. “A MBSE Approach to MDAO Systems for the Development of Complex Products”. In: *AIAA Aviation Forum*. Reno, Nevada, June 2020. DOI: 10.2514/6.2020-3150.

[309] H. Wagner and C. Zuccaro. “Collaboration between System Architect and Simulation Expert”. In: *2022 IEEE International Symposium on Systems Engineering (ISSE)*. IEEE, Oct. 2022, pp. 1–8. DOI: 10.1109/isse54508.2022.10005417.

[310] R. Braun et al. “Hopsan: An Open-Source Tool for Rapid Modelling and Simulation of Fluid and Mechatronic Systems”. In: *BATH/ASME 2020 Symposium on Fluid Power and Motion Control*. FPMC2020. American Society of Mechanical Engineers, Sept. 2020. DOI: 10.1115/fpmc2020-2796.

[311] C. Jouannet, K. Amadori, and A. Papageorgiou. “Technology Assessment for Unmanned AEW/ISR Platform”. In: *AIAA SCITECH 2022 Forum*. American Institute of Aeronautics and Astronautics, Jan. 2022. DOI: 10.2514/6.2022-0129.

[312] S. Reitenbach et al. “Collaborative Aircraft Engine Preliminary Design using a Virtual Engine Platform, Part A: Architecture and Methodology”. In: *AIAA Scitech 2020 Forum*. American Institute of Aeronautics and Astronautics, Jan. 2020. DOI: 10.2514/6.2020-0867.

[313] J.M. Parr. “Improvement Criteria for Constraint Handling and Multiobjective Optimization”. PhD thesis. University of Southampton, Feb. 2013.

[314] S.K. Lam, A. Pitrou, and S. Seibert. “Numba: a LLVM-based Python JIT compiler”. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. SC15. ACM, Nov. 2015. DOI: 10.1145/2833157.2833162.

[315] C. Audet et al. “Performance indicators in multiobjective optimization”. In: *European Journal of Operational Research* 292.2 (July 2021), pp. 397–422. DOI: 10.1016/j.ejor.2020.11.016.

A

Bayesian Optimization: Equations and Infill Search Procedure

This appendix presents several equations and procedures related to Gaussian Process (GP) models and Bayesian Optimization (BO).

A.1. Gaussian Process Models

For a given design point $\boldsymbol{x}$, GP models, also known as Kriging models, provide both the most likely value $\hat{y}(\boldsymbol{x})$ of that point and a standard deviation $\hat{\sigma}_y(\boldsymbol{x})$, which represents how confident the model is about the value prediction at $\boldsymbol{x}$. [167]. A short explanation of how GP models calculate $\hat{y}$ and $\hat{\sigma}_y$ for noiseless training data follows, for more details the reader is referred to [49, 167].

A GP is a random process where any point in the design space is a random variable and where the join distribution of these variables is Gaussian [167]:

$$p\left(\boldsymbol{y}\middle|\boldsymbol{X}\right) = \mathcal{N}\left(\boldsymbol{y}\middle|\boldsymbol{\mu},\boldsymbol{K}\right), \tag{A.1}$$

where $\boldsymbol{y}$ is the vector of training values at design vectors $\boldsymbol{X}$. $\boldsymbol{\mu}$ represents the mean function, which is commonly set to 0 for black-box functions (this is referred to as "simple Kriging"). $\boldsymbol{K}$ is the kernel matrix, with values $K_{ij} = \kappa\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right)$ defined by the positive definite kernel function κ (also known as covariance function). The kernel function κ represents the covariance of the design points and their function values:

$$\kappa\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = \mathrm{cov}\left(y_i(\boldsymbol{x}_i), y_j(\boldsymbol{x}_j)\middle|\boldsymbol{\theta}\right), \tag{A.2}$$

where $\boldsymbol{\theta}$ are the hyperparameters of the kernel function. The above equation means that if two points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are located closer together, their associated function values $y_i(\boldsymbol{x}_i)$ and $y_j(\boldsymbol{x}_j)$ also lie closer together. Kernel functions are based on distances.

For a continuous space this for example can be the Euclidean distance, however a distance metric can also be defined for discrete and/or hierarchical spaces [174]. Kernel functions rely on hyperparameters $\boldsymbol{\theta}$, chosen to maximize the likelihood that the GP fits the training values [49].

An output value $\hat{y}(\boldsymbol{x})$ is predicted by finding the value that maximizes the likelihood of the GP model if the point would be part of the training set:

$$p\left(\hat{y}|\boldsymbol{x}\right) = \mathcal{N}\left(\begin{matrix}\boldsymbol{y}\\ \hat{y}\end{matrix}\middle|\boldsymbol{0}, \begin{matrix}\boldsymbol{K} & \boldsymbol{\kappa}_*\\ \boldsymbol{\kappa}_*^T & 1\end{matrix}\right), \tag{A.3}$$

where $\boldsymbol{\kappa}_*$ is the vector containing $\kappa(\boldsymbol{x}, \boldsymbol{x}_i)$ for every point $\boldsymbol{x}_i$ in the training set. Solving the above equation for the maximum likelihood is beyond the scope of this appendix, however details are provided in [49, 167]. Doing so results in the equation for the mean value prediction:

$$\hat{y}(\boldsymbol{x}) = \boldsymbol{\kappa}_*^T \boldsymbol{K}^{-1} \boldsymbol{y}, \tag{A.4}$$

and the equation for predicting the associated standard deviation:

$$\hat{\sigma}_y(\boldsymbol{x}) = 1 - \boldsymbol{\kappa}_*^T \boldsymbol{K}^{-1} \boldsymbol{\kappa}_*. \tag{A.5}$$

A.2. Bayesian Optimization Infill Search Procedure

Selecting the best infill points in a BO algorithm is an optimization problem itself, sharing the design space with the SAO problem. To deal with mixed-discrete hierarchical variables, the following sequential optimization procedure is applied:

1. Use NSGA-II [155] to search the mixed-discrete, hierarchical design space, solving the multi-objective infill problem:

$$\begin{aligned} &\text{minimize} && f_{\text{infill,m}}(\boldsymbol{x}_d, \boldsymbol{x}_c)\\ &\text{w.r.t.} && \boldsymbol{x}_d, \boldsymbol{x}_c\\ &\text{subject to} && g_{\text{infill,k}}(\boldsymbol{x}) \leq 0, \end{aligned} \tag{A.6}$$

   where $f_{\text{infill,m}}$ represents infill criterion m (see Sections A.3 and A.4), $\boldsymbol{x}_d$ and $\boldsymbol{x}_c$ represent the discrete and continuous design variables, and $g_{\text{infill,k}}(\boldsymbol{x})$ represents the constraint function k (see Section A.5). Infill objectives $f_{\text{infill,m}}$ are normalized and inverted such that they become minimization objectives.
2. Select n_{batch} points $\boldsymbol{x}_{\text{sel}}$ from the resulting Pareto front:
   - If $n_{\text{batch}} = 1$, select a random point.
   - If $n_{\text{batch}} > 1$, select the points with the lowest crowding distance (as used in NSGA-II [155]).
3. For each selected point $\boldsymbol{x}_{sel,i}$, improve the active continuous variables by solving following single-objective problem using SLSQP[1]:

$$\begin{aligned} &\text{minimize} && f_{\text{impr}}(\boldsymbol{x})\\ &\text{w.r.t.} && \boldsymbol{x}_c \cap \delta\left(\boldsymbol{x}_{\text{sel,i}}\right)\\ &\text{subject to} && g_{\text{infill,k}}(\boldsymbol{x}) \leq 0, \end{aligned} \tag{A.7}$$

[1] https://docs.scipy.org/doc/scipy/reference/optimize.minimize-slsqp.html

where $\boldsymbol{x}_c \cap \delta\left(\boldsymbol{x}_{\text{sel,i}}\right)$ represents the active continuous design variables at point $\boldsymbol{x}_{\text{sel,i}}$, and f_{impr} the scalar improvement objective:

$$\begin{aligned}
\Delta f_{\text{infill,m}}(\boldsymbol{x}) &= f_{\text{infill,m}}(\boldsymbol{x}) - f_{\text{infill,m}}(\boldsymbol{x}_{sel,i}) \\
f_{\text{deviation}}(\boldsymbol{x}) &= 100\left(\max_{\text{m}} \Delta f_{\text{infill,m}}(\boldsymbol{x}) - \min_{\text{m}} \Delta f_{\text{infill,m}}(\boldsymbol{x})\right)^2 \\
f_{\text{impr}}(\boldsymbol{x}) &= \sum_{\text{m}} \left(\Delta f_{\text{infill,m}}(\boldsymbol{x})\right) + f_{\text{deviation}}(\boldsymbol{x}).
\end{aligned} \tag{A.8}$$

This objective promotes improvement in direction of the negative unit vector, so that all underlying infill objectives are improved simultaneously while not deviating too much from the original $x_{\text{sel,i}}$ to maintain batch diversity.

A.3. SINGLE-OBJECTIVE INFILL CRITERIA

For single-objective SAO problems, an infill ensemble combining the following infill criteria is used:

1. Lower Confidence Bound (LCB) [219], given by:

$$f_{\text{infill},1}(\boldsymbol{x}) = \hat{f}(\boldsymbol{x}) - \alpha\, \hat{\sigma}_f(\boldsymbol{x}), \tag{A.9}$$

with α being a scaling parameter of which lower (higher) values lead to more exploitation (exploration). By default, $\alpha = 2$ is used.

2. Expected Improvement (EI) [142], given by:

$$\begin{aligned}
f_{\text{infill},2}(\boldsymbol{x}) &= 1 - \text{EI}(\boldsymbol{x}), \\
\text{EI}(\boldsymbol{x}) &= \left(f^* - \hat{f}(\boldsymbol{x})\right)\Phi\left(\frac{f^* - \hat{f}(\boldsymbol{x})}{\hat{\sigma}_f(\boldsymbol{x})}\right) + \hat{\sigma}_f(\boldsymbol{x})\phi\left(\frac{f^* - \hat{f}(\boldsymbol{x})}{\hat{\sigma}_f(\boldsymbol{x})}\right),
\end{aligned} \tag{A.10}$$

where f^* is the current best value of f, and Φ and ϕ are the cumulative and probability distribution functions of the normal distribution, respectively.

3. Probability of Improvement (PoI) [220], given by:

$$\begin{aligned}
f_{\text{infill},3}(\boldsymbol{x}) &= 1 - \text{PoI}(\boldsymbol{x}), \\
\text{PoI}(\boldsymbol{x}) &= \Phi\left(\frac{f^* - \hat{f}(\boldsymbol{x})}{\hat{\sigma}_f(\boldsymbol{x})}\right),
\end{aligned} \tag{A.11}$$

A.4. MULTI-OBJECTIVE INFILL CRITERIA

For multi-objective SAO problems, an infill ensemble combining the following infill criteria is used:

1. Minimum Probability of Improvement (MPoI) [221], given by:

$$\begin{aligned}
f_{\text{infill},1}(\boldsymbol{x}) &= 1 - \text{MPoI}(\boldsymbol{x}), \\
\text{MPoI}(\boldsymbol{x}) &= \min_i \left(1 - \prod_m \Phi\left(\frac{\hat{f}_m(\boldsymbol{x}) - f^*_{\text{i,m}}}{\hat{\sigma}_{\text{f,m}}(\boldsymbol{x})}\right)\right),
\end{aligned} \tag{A.12}$$

where $f^*_{\text{i,m}}$ represents objective m of point i in the Pareto front.

2. Minimum Euclidean PoI (MEPoI) [10, 313], given by:

$$\begin{aligned} f_{\text{infill},2}(\boldsymbol{x}) &= 1 - \text{EM}(\boldsymbol{x})\text{MPoI}(\boldsymbol{x}), \\ \text{EM}(\boldsymbol{x}) &= \begin{cases} \min\limits_{i} \sqrt{\sum\limits_{m} \left(\hat{f}_m(\boldsymbol{x}) - f^*_{\text{i,m}}\right)^2} & \text{if } \text{MPoI}(\boldsymbol{x}) > 50\% \\ 0 & \text{else} \end{cases}, \end{aligned} \tag{A.13}$$

where EM represents the Euclidean moment [313].

A.5. Constraint Handling

Inequality design constraints are handled by the infill constraint functions $g_{\text{infill},k}(\boldsymbol{x})$ for each design constraint k. The following options can be used as infill constraint functions:

- Constraint mean prediction [171], given by:

$$g_{\text{infill,k}}(\boldsymbol{x}) = \hat{g}_k(\boldsymbol{x}), \tag{A.14}$$

  where $\hat{g}_k$ is the mean prediction of the GP model trained for design constraint k.

- Probability of Feasibility (PoF) [222], given by:

$$\begin{aligned} g_{\text{infill,k}}(\boldsymbol{x}) &= \text{PoF}_{\text{min}} - \text{PoF}_k(\boldsymbol{x}), \\ \text{PoF}_k(\boldsymbol{x}) &= \Phi\left(\frac{-\hat{g}_k(\boldsymbol{x})}{\hat{\sigma}_{\text{g,k}}(\boldsymbol{x})}\right). \end{aligned} \tag{A.15}$$

By default, constraint function mean prediction is used.

B
DSG Selection Choice Encoders

This appendix provides more details about the selection choice encoders, as introduced in Section 3.1.1. The selection choice encoding algorithms implement the following operations, given a Design Space Graph (DSG):

- Encode selection choices into a set of discrete design variables.

  Selection choice nodes are sorted by topological order as seen from the start nodes, resulting in that selection choices that are taken earlier in the derivation process are moved to the left of the design vector.

- Determine which design variables are forced.

  Design variables are forced if their value can be fully determined by the values of other design variables (see Section 3.1.1 for more details), and as such they can be ignored by the optimization algorithm as they do not add any extra information to the design vector.

- For a given design vector x:

  - correct the design vector if needed;
  - return the corrected design vector x;
  - return associated activeness information δ; and
  - optionally return the associated DSG instance.

- Enumerate the valid design vectors $x_{\text{valid,discr}}$ and associated activeness information $\delta_{\text{valid,discr}}$.

  Note that providing this and the following functionalities might not be possible or feasible due to memory or time limits.

- Calculate the number of valid design vectors $n_{\text{valid,discr}}$.

- Enumerate the node existence status vectors associated to the valid design vectors.

  Each node in the DSG is assigned an element in the vector: (0) node is not included, (1) node is confirmed, and (2) node is removed. Node existence status vectors therefore allow the DSG to check for which combinations of selection choice design variables a set of nodes are included.

- Get all unique node existence status vectors for a set of nodes.

  This information is for example used to define node existence scenarios in Connection Choice Formulations (CCFs), see Section 3.1.2 and Appendix C for more details.

The Python implementation of the selection choice encoders is available open-source as part of the DSG in the ADSG CORE[1] package. The base for the selection choice encoders is the `HierarchyAnalyzerBase` class. The following sections present the fast and complete selection choice encoders in more details.

B.1. FAST ENCODER

The fast encoder does not implement valid design vector and associated existence status vector enumeration. It therefore provides the DSG with less information about the design space behavior, however encodes a DSG into a set of design variables practically instantly compared to the complete encoder. The fast encoder is implemented in the `FastHierarchyAnalyzer` class. It cannot detect forced choices, except those coming from linked choice constraints (see Section 3.1.3). Decoding a design vector x into a DSG instance by the fast encoder is done as follows:

1. Start from the DSG representing the architecture design space.
2. Repeat as long as there are active selection choice nodes:
   (a) Request the next active selection choice node.
   (b) Get the option index as selected by the associated design variable in x.
   (c) Resolve the selection choice with the selected option node.
   (d) If the option node was not available, quit the loop.
   (e) Otherwise, note the taken option index, thereby also noting which choices were not taken and therefore inactive.
3. If the loop was quit due to an unavailable option, or if the resulting DSG instance is not feasible (e.g. due to infeasible connection choices or infeasible incompatibility constraints): mark the design vector as invalid, generate a new design vector by iterating over neighboring points, and start from step 1.
4. A feasible DSG instance is found, remember the associated feasible design vector for the original input design vector. Return the (corrected) design vector, associated activeness information, and generated DSG instance.

[1] https://adsg-core.readthedocs.io/

Correction is thus done by a trail-and-error approach, and there is no mechanism to correct a design vector without generating the associated DSG instance. This is why the fast decoder is slightly slower for correction compared to the complete encoder, as shown in Section 3.6.

B.2. Complete Encoder

The complete encoder is able to fully enumerate all valid design vectors $x_{\text{valid,discr}}$, associated activeness information $\delta_{\text{valid,discr}}$, and associated node status vectors. The complete encoder starts by constructing an influence matrix: a matrix defining for all selection choice nodes and their option nodes how they influence each other and all other generic nodes through derivation and incompatibility edges.

The influence matrix is a square matrix where each row/column represents a specific node in the DSG. It contains initial node statuses on the diagonal, and node influences (derivation or removal) on the off-diagonal. Conceptually, the influence matrix aggregates derivation and incompatibility edges in the DSG, so that for each option node, all downstream activation or removal influences can be determined. This enables for a given combination of selected option nodes to determine which selection choices were active, which option nodes were available, and which nodes are confirmed in the associated final architecture graph. Selection choice activation and option node availability information can also be derived for partial architectures (architectures where some but not all of the selection choices have options assigned). This enables enumeration of all possible combinations of selection choice options, while keeping track of selection choice activeness and node existence.

Table B.1 shows the influence for the DSG shown in Figure 3.9. It shows that C1, N1, N2 and N3 are initially active (1 on the diagonal) and C2 and the other nodes not (0 on the diagonal). Nodes N4, N5, N6 and N13 are option nodes of C1 and the off-diagonal influences in their respective rows indicate the nodes (in columns) which they influence if they are selected as option for C1. An off-diagonal 1 (green) indicates that the node in the row activates the node in the column if selected; and off-diagonal 2 (red) indicates removal. For example consider the interaction between C1 and C2: selecting N4 for C1 activates C2 and leaves all its option nodes available, however selecting N13 activates C2 and removes N8 as option from C2. This behavior is consistent with Table 3.1.

Encoding selection choices into discrete design variables is done by defining one design variable per selection choice, removing options that never lead to feasible DSG instances, and detecting which design variables are forced choices.

Two implementations of the complete encoder are available: the brute force encoder and the declarative lazy encoder. The differ in the ways they perform design vector correction, determine forced choices, and enumerate valid design vectors. The brute force encoder enumerates all valid design vectors a-priori, resulting in a simpler implementation at the cost of more memory usage. The code has been structured, however, to allow the implementation of new complete encoders by subclassing the `SelectionChoiceEncoder` base class. The brute force encoder can be used to verify that the behavior of newly implemented encoders is correct, by ensuring that all encoder operations return the same results.

Table B.1: Influence matrix for the DSG shown in Figure 3.9. On the diagonal, green nodes (1) indicate activated nodes. Off-diagonal, green nodes (1) indicate that the node in the row activates/selects the node in the column, and red nodes (2) indicate deactivation/removal.

	C1	N4	N5	N6	N13	C2	N8	N11	N1	N2	N3	N7	N9	N10
C1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
N4	0	0	0	0	0	1	0	0	0	0	0	1	0	0
N5	0	0	0	1	2	1	1	0	0	0	0	1	1	1
N6	0	0	0	0	2	0	1	0	0	0	0	0	1	1
N13	0	0	2	2	0	1	2	0	0	0	0	1	2	2
C2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N8	0	0	0	0	2	0	0	0	0	0	0	0	1	1
N11	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
N2	0	0	0	0	0	0	0	0	0	1	0	0	0	0
N3	0	0	0	0	0	0	0	0	0	0	1	0	0	0
N7	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N10	0	0	0	0	0	0	0	0	0	0	0	0	0	0

B.2.1. BRUTE FORCE ENCODER

The `BruteForceSelectionChoiceEncoder` class implements the brute force encoder. Valid design vector enumeration is done by a branching algorithm, which keeps track of node status vectors as modified by taking choices, and creates new branches on remaining active selection choices. Finding the closest-matching design vector index for a given design vector is done by first checking if there is an exact match (ignoring forced and inactive design variables). If no exact match was found, the closest matching design vector by euclidean distance is selected instead.

B.2.2. DECLARATIVE LAZY ENCODER

The declarative lazy encoder is implemented in the `HierarchyAnalyzer` class, and provides all the operations requested for a complete encoder without requiring the a-priori enumeration of all valid design vectors. It does so by formulating selection scenarios for each selection choice: each selection scenario represents a unique combination of previously-selected option nodes of other selection choices, and specifies which remaining available options are available for the selection choice, and what their influences are on downstream node statuses. The set of all selection scenarios for all selection choices together declaratively specify all the valid design vectors, without the need for upfront design vector enumeration: hence the name "declarative lazy".

In the current implementation, scenario sets are strategically merged such that in the end only one or more independent scenario sets and associated dependent scenario sets remain. The independent scenario sets do not depend on other selection choices, and the dependent sets only depend on the selection choices of the associated independent scenario set (i.e. they are not mutually dependent). Each time two scenario sets are merged, however, the valid selection choice options between the merged scenario sets are enumerated, which increases memory usage and reduces the advantage over the brute force encoder. The fact that only one level of scenario dependency is currently supported therefore represents a limitation, one which could be relieved in the future by improved complete encoder implementations.

Finding the closest-matching design vector index for a given design vector is done by first checking if the independent scenario sets contain the requested design variable values, and correcting to the closest matching (by euclidean distance) partial design vector if needed. Once the independent scenario sets have been selected, the dependent scenario sets are parsed and partial design vectors are corrected if needed. All other encoder operations progress in a similar fashion from independent to dependent scenario sets. For example, when enumerating all valid design vectors (if that is requested), first for each independent scenario set and associated dependent scenario sets the valid (partial) design vectors are generated, which are then combined by Cartesian product (because they are independent) into the full matrix of valid design vectors.

B.2.3. Design Vector Decoding

Decoding a design vector x into a DSG instance by the complete encoders is done as follows:

1. Find the closest-matching valid design vector index and associated design vector x. In that process, the originally provided design vector is corrected if no exact match is found. Activeness information δ is obtained from the (corrected) design vector.

   Optionally, the set in which to search for design valid design vectors can be restricted, for example to exclude infeasible DSG instances.

   If only correction is requested (i.e. the associated DSG instance is not needed), the process stops here, returning the (corrected) design vector and design vector index, and associated activeness information.
2. Start from the DSG representing the architecture design space.
3. Repeat as long as there are active selection choice nodes:
   (a) request the next active selection choice node;
   (b) get the option index as selected by the associated design variable in x; and
   (c) resolve the selection choice with the selected option node.
4. If the resulting DSG instance is not feasible (e.g. due to infeasible connection choices or infeasible incompatibility constraints): mark the design vector as invalid and restart the decoding process.
5. A feasible DSG instance is found. Return the (corrected) design vector and design vector index, the associated activeness information, and the generated DSG instance.

C

DSG Connection Choice Encoders

This appendix presents the database of connection choice encoders, as introduced in Section 3.1.2. Connection choice encoders always act on one connection choice at a time, each of which are defined by a Connection Choice Formulations (CCF), consisting of:

- Source and target connector nodes, which are each defined by:
  - an allowable number of connections: either as a list of integers, or as a lower bound and optional upper bound (inclusive); and
  - a flag dictating whether from/to that node parallel connections to/from some other node can be established.
- An optional list of excluded edges, defined by pairs of nodes or pairs of node indices.
- An optional list of node existence scenarios, which may define any of the following:
  - for each source and/or target node, whether they exist or not;
  - for each source and/or target node, the number of connections they allow (as a list of integers); and/or
  - for source or target nodes, an upper bound on the number of connections they may allow.

  Note that if no existence scenarios are provided, the default existence scenario is assumed to be the only scenario. This scenario assumes all nodes exist and does not override any allowable connection settings.

  For each node existence scenario, an "effective CCF" can be formulated which represents the CCF with node existence and/or allowed number of connections modified.

- An optional upper bound on the number of parallel connections that may be established between pairs of nodes.

A given connection pattern is defined by a $n_{\text{src}} \times n_{\text{tgt}}$ (number of source and target nodes, respectively) connection matrix M, of which element i, j represents the number of connections from source i to target j. Valid connection matrices can be enumerated for each existence scenario of a CCF, which are represented in a $n_{\text{src}} \times n_{\text{tgt}} \times n_{\text{M,valid}}$ matrix $M_{\text{all,valid}}$, where $n_{\text{M,valid}}$ is the number of valid connection matrices. Note that it might not be possible or feasible to determine $M_{\text{all,valid}}$ due to memory or time limits. Each connection choice encoder implements the following operations:

- Encode a CCF into a set of discrete design variables x.
- Calculate the IR_d and Dcorr metrics as introduced in Section 3.1.2.
- For a given design vector $\boldsymbol{x}$ and node existence scenario:
  - check whether it is a valid design vector;
  - if not, correct the design vector;
  - return the corrected design vector x;
  - return the associated connection matrix M; and
  - return activeness information δ for the corrected x.
- Enumerate all valid design vectors $x_{\text{valid,discr}}$ and associated activeness information $\delta_{\text{valid,discr}}$ for each node existence scenario. Note that this might not be possible or feasible due to memory or time limits.

The Python implementation of connection choice encoders is available open-source as part of the DSG in the ADSG Core[1] package. Everything related to connection choice encoding is implemented in the `adsg_core.optimization.assign_enc` module. CCFs are implemented by the `MatrixGenSettings` class; matrix enumeration is implemented by the `AggregateAssignmentMatrixGenerator` class. The connection matrix enumeration code uses Numba [314] and caches results to speed up execution.

The following sections present the four different classes of connection choice encoders in the order of decreasing preference (see also Table 3.5). Section C.5 presents the algorithm for selecting an encoder for a given CCF.

C.1. Pattern-specific Encoders

Each pattern-specific encoder is developed to optimally (in terms of IR_d and Dcorr) encode a specific architecture decision pattern [128], as listed in Table 3.4. Compared to generic encoders, pattern encoders are the fastest in terms of encoding, decoding, and correcting time, and require the least amount of memory to do so (see Table 3.1.2). However, the pattern-specific encoders only cover relatively specific CCFs. Each pattern-specific encoder implements a function to check whether the encoder is compatible with a given effective CCF. An encoder can then be used to encode a CCF if the encoder is compatible with (the transpose of) the effective CCFs of all node

[1] https://adsg-core.readthedocs.io/

existence scenarios. The transpose effective CCFs are checked, because this allows the pattern-specific encoders to be implemented in only one "direction", reducing code duplication and the implementation effort. A transpose CCF represents a CCF with source and target nodes swapped. If a pattern-specific encoder is used for a transposed CCF, the connection matrices resulting from decoded design vectors are transposed before returned.

The connection patterns as listed in Table 3.4 can exist in different configurations, and some patterns are actually equal to or specializations of other patterns:

- bijective assigning is the same as partitioning if source nodes allow $0..*$ connections;
- injective assigning is the same as downselecting; and
- surjective assigning is the same as covering partitioning if the source nodes allow $0..*$ connections.

Table 3.4 also shows that none of the patterns are transposed versions of each other, showing that source and target node definitions can be freely exchanged. Also note that any CCF where not both sources and targets allow parallel connections are effectively non-parallel. Table C.1 lists all implemented pattern-specific encoders, which configurations of source and target nodes they are compatible with, which patterns of Table 3.4 are matched for each encoder, how the patterns are encoded into discrete design variables, and how design vectors are corrected if needed.

C.2. EAGER ENCODERS

Eager encoders can encode any CCF, and to do so require the a-priori enumeration of $M_{\text{all,valid}}$. Encoding is then done as follows:

1. Encoder-specific: associate each M to a unique design vector, for each node existence scenario.
2. Optionally normalize the design vectors:
   - move the design variables such that their lower bound is 0;
   - remove design variables with less than two unique values; and
   - optionally remove value gaps (e.g. if the unique values for a given design variable are $0,1,3$, modify it such that the unique values are $0,1,2$).
3. Derive discrete design variables from the matrix of design vectors.
4. Merge discrete design variables derived for the different node existence scenarios, such that the upper bounds are large enough.

Decoding a design vector x for a given node existence scenario is done as follows:

1. Correct the upper bounds of the design vector values to be consistent with the design variables of the provided node existence scenario.
2. Check if the design vector matches an encoded design vector (ignoring inactive values). If a match is found, return the associated M.

Table C.1: Overview of pattern-specific encoders and which architecture decision patterns, see Table 3.4, they are compatible with. n, m and k are non-negative integers. How patterns are encoded into design variables is reported as the number of discrete design variables n_{x_d} and the number of options for design variable j: N_j. $n_{\parallel,\max}$ represents the maximum number of parallel connections; $n_{\parallel,\max} = 1$ if parallel connections are not allowed.

Encoder	Configuration	Source nodes	Target nodes	Constraints	Compatible patterns	n_{x_d}	N_j	Correction
Combining		$1 @ 1$	$m @ 0,1$		Combining	1	m	Clipping
Combining	Collapsed	$1 @ 0..* \parallel$	$1 @ 0..* \parallel$		Combining	1	m	Clipping
Unordered combining	With replacement	$1 @ n \parallel$	$m @ 0..* \parallel$		Unordered combining	$n+m-2$	2	Random
Unordered combining		$1 @ n$	$m @ 0,1$	$n \leq m$	Combining, unordered non-replacing combining	$n-1$	2	Random
Assigning		$n @ 0..*$	$m @ 0..*$		Assigning	$n \cdot m$	2	Random
Assigning	Surjective	$n @ k..*$	$m @ 1..*$		Surjective assigning, covering partitioning	$n \cdot m$	2	Random
Assigning	Parallel	$n @ 0..* \parallel$	$m @ 0..* \parallel$		Parallel assigning	$n \cdot m$	$n_{\parallel,\max}+1$	Random
Assigning	Parallel surjective	$n @ k..* \parallel$	$m @ 1..* \parallel$		Parallel surjective assigning	$n \cdot m$	$n_{\parallel,\max}+1$	Random
Partitioning		$n @ k..*$	$m @ 1$	$k \cdot n \leq m$	Partitioning, bijective assigning	m	n	Random
Partitioning	Downselecting	$n @ k..*$	$m @ 0,1$	$k \cdot n \leq m$	Downselecting, injective assigning	m	$n+1$	Random
Connecting		$n @ 0..*$	$n @ 0..*$	(i,j) if $i \geq j$	Connecting	$(n \cdot (n-1))/2$	2	Not needed
Connecting	Directed	$n @ 0..*$	$n @ 0..*$	(i,j) if $i = j$	Directed connecting	$n \cdot (n-1)$	2	Not needed
Permuting		$n @ 1$	$n @ 1$		Permuting	$n-1$	$n-j$	Not needed

3. If no match is found, correct the design vector: starting from the left-side of x, progressively filter the available design vectors that match the partial design vector. If no design vectors remain, modify the latest variable to an available option and continue.

   This is a greedy correction algorithm, as design variables are corrected locally as x is being processed. Alternatives would be distance-based correction, where the closest matching design vector is chosen according to some distance metric, however tests showed that greedy correction achieved better optimization results and resulted in faster correction times.

Two types of greedy encoders are implemented: the direct matrix encoder and a set of grouping encoders. The direct matrix encoder (`DirectMatrixEncoder` class) simply flattens the enumerated connection matrices into design vectors. Connection matrix M element i, j (0-indexed) is therefore directly represented by design variable $k = i \cdot n_{\text{tgt}} + j$. Design vector normalization is applied, including value gap removal.

Grouping encoders (`GroupedEncoder` base class) work by repeatedly separating a grouping matrix G into groups, until each group represents one unique design vector. The process is as follows:

- For each column j in G:
  - For each group at level j:
    - Skip group if the group size is 1 (no need to continue defining sub-grouping), and set current and subsequent design variables to inactive.
    - Get unique values within the group.
    - Skip group if there is only 1 unique value, set current design variables to inactive.
    - Assign design variable values from unique value indices, and define a new group for each unique value for level $j+1$.
- Remove design variables that are always inactive.
- Optionally convert the design variable values to some other base (e.g. base 2).

  This operation might increase Dcorr (wanted) at the cost of increasing IR_d (unwanted).

Table C.2 shows an example of encoding a grouping matrix G into design vectors x. As can be seen, groups are progressively created while processing the columns of G, until all groups have size 1. Column 2 does not create any new groups compared to column 2, resulting in a completely inactive x_2, which will subsequently be removed from the design vectors. It can also be seen that once a group reaches a size of 1, all subsequent design variables in that group are inactive. Finally, encoded values within a group are normalized to 0, and gaps are removed.

Table C.3 lists the available grouping encoders. Note that more amount-first grouping encoders could be defined, for example by also involving the connection indices for grouping after grouping by amount of connections, however these did not prove effective and are therefore not used in practice.

Table C.2: Illustration of the grouping process used by grouping encoders. G_j represents column j of the grouping matrix G. x_j represents encoded values for design variable j; empty cells represent inactive design variables. Horizontal lines represent group delimiters, shown from the column where they are created.

Grouping matrix G						Encoded design vectors x					
G_0	G_1	G_2	G_3	G_4	G_5	x_0	x_1	x_2	x_3	x_4	x_5
0	0	1	0	0	1	0	0		0		0
0	0	1	0	0	2	0	0		0		1
0	0	1	0	0	4	0	0		0		2
0	0	1	2	0	0	0	0		1	0	
0	0	1	2	1	1	0	0		1	1	
0	0	1	3	0	4	0	0		2	0	
0	0	1	3	1	0	0	0		2	1	
0	1	2	0	0	1	0	1				
2	1	2	1	1	5	1					

Table C.3: Overview of eager grouping encoders.

Encoder	Variation	Group by
Element grouping		Flattened connection matrices M
Connection index grouping	By source	Connection indices for each source node
	By target	Connection indices for each target node
	By source, base 2	Connection indices for each source node
	By target, base 2	Connection indices for each target node
Amount-first grouping	Total amount	By amount of connections, then ordinal
	Source amount	By amount of connections per source node, then ordinal
	Target amount	By amount of connections per target node, then ordinal

C.3. LAZY ENCODERS

Lazy encoders directly use the effective CCF to define design variables, preventing the need to enumerate $M_{\text{all,valid}}$. Decoding a design vector x for a given node existence scenario is then done as follows:

1. Encoder-specific: construct a connection matrix M from the design vector x.
2. If no M could be constructed, or if the resulting M is not valid according to the constraints specified in the effective CCF, correct the design vector.

   The design vector is corrected by generating adjacent design vectors, by looping over the Cartesian product of available design variable delta's and adding these to the original x. Each generated x is checked for validity against the effective CCF, until a valid x is found, which then becomes the corrected x.

Correction is based on a trial-and-error approach and therefore might take a relatively long time, especially if the encoder has a high IR_d. Following lazy encoders are available:

- Direct matrix encoder (`LazyDirectMatrixEncoder` class): this encoder works similarly as the eager direct matrix encoder, in that it defines a design variable for each source and target node pair, with the number of maximum parallel connections taken as N_j for each design variable. Depending on the CCF, however, many combinations of connections might be invalid, and therefore this encoder potentially defines many design variable (options) that might not yield valid matrices.
- Connection index encoder (`LazyConnIdxMatrixEncoder` class): this encoder is similar to the eager connection index encoder, in that is defines design variables that represent the connection index, either source-node-wise or target-node-wise. It does so by enumerating valid connection matrices that represent cases with the most connections. The encoding is less efficient than the eager variant, however, because assumptions have to be made about the valid number of connections based on the CCF. These assumptions have to be conservative in nature, because it has to be ensured that all matrices in $M_{\text{all,valid}}$ can be encoded.

  This encoder is used in the following variants:

  - one design variable ordinally encoding the number of available valid connection matrices;
  - several design variables encoding the valid connection matrices using the grouping procedure of the grouping eager encoders; or
  - design variables encoding the connection indices in the valid connection matrices, optionally converted to base-2.
- Amount-first encoder (`LazyAmountFirstEncoder` class): this encoder is similar to the eager amount-first encoder, in that first design variables are defined to encode the number of connections, and subsequent design variables encode the connection indices. This encoder is less efficient than the eager variant for the same reason as for the connection index encoder.

  This encoder is used in two variants: with one design variable ordinally mapping the unique number of total connections, or with one design variable for each source and target node specifying the number of connections per node.

C.4. ORDINAL ENCODERS

Ordinal encoders simply map each M in $M_{\text{all,valid}}$ to an index, and either define one variable with $n_{\text{M,valid}}$ options, or encode the indices in some base-k system. For these encoders, the IR_d is usually not too bad (if one variable is defined, $\text{IR}_d = 1$ even). However, Dcorr is usually around 0%, because the encoded indices do not provide any information about the associated connection matrix M. Therefore, ordinal encoders should only be used as a last resort. Table C.4 lists the available ordinal encoders. In practice, the ordinal encode without base conversion, and base-converted ordinal encoders with $k = 2, 3, 4$ are used. Higher values of k quickly approach the behavior of the ordinal encoder without base conversion, and are therefore not needed.

Table C.4: Overview of ordinal encoders, listing the number of discrete design variables n_{x_d} and number of options per discrete design variable j: N_j.

Base	n_{x_d}	N_j
None	1	$n_{\text{M,valid}}$
k	$\lceil \log_k \left(n_{\text{M,valid}} \right) \rceil$	k

C.5. AUTOMATIC ENCODER SELECTION

The selection algorithm is implemented in the `EncoderSelector` class, with the goal of automatically selecting an encoder that maximizes Dcorr and minimizes IR_d for a given CCF. The selection algorithm works by iteratively trying to create sets of encoders and select the best encoder from the created set for a given CCF.

The following process is used to try to instantiate one encoder:

1. Create the encoder and encode design variables, automatically stopping the process if:
   - a memory overflow error is encountered (in Python, memory overflow errors are recoverable, as the encoder object and associated memory usage is destroyed in the process of catching the error);
   - some time limit is exceeded (250 ms in the current implementation); or
   - a high imputation ratio is detected (100 in the current implementation) during the recursive grouping process of eager grouping encoders.
2. Calculate the IR_d metric.
3. Calculate the Dcorr metric, if $\text{IR}_d \leq 100$ for eager encoders or $\text{IR}_d \leq 10$ for lazy encoders. As calculating Dcorr involves sampling design vectors and associated connection matrices, this process can take a long time (for example if decoding or correction is slow), and therefore also here a time limit is used (the same limit as used for encoding).
4. If a perfect encoder is found ($\text{IR}_d = 1$ and Dcorr $\geq 95\%$), no further encoders are created in order to save time.

The following process is used to select the best encoder from a set of created encoders:

1. Divide the IR_d - Dcorr landscape in priority areas, as defined in Table C.5.
2. Find the highest-priority area that contains at least one encoder.
3. Within the selected priority area, select the encoder with the highest Dcorr. If there are multiple encoders that share the highest Dcorr, select the encoder with the lowest IR_d within that set.

The overall selection process is then as follows:

1. Enumerate valid connection matrices $M_{\text{all,valid}}$ for each node existence scenario. The result of this is cached, so that the enumeration does not have to be repeated for each tried encoder.

Table C.5: Priority areas in the IR_d - Dcorr landscape for selecting the best encoder within a set of encoders. Lower numbers have a higher priority for selection. Areas of lower priority do not include areas already covered by higher priorities.

Dcorr			IR_d		
$\downarrow$	$= 1$	≤ 10	≤ 40	≤ 100	> 100
$\geq 70\%$	1	2	9	13	17
$\geq 35\%$	3	4	10	14	18
$> 0\%$	5	6	11	15	19
$= 0\%$	7	8	12	16	20

2. Try creating pattern-specific encoders. If any compatible pattern-specific encoder is created, select the best from that list, considering only the first 5 priority areas. If successful, use the selected encoder and quit the process.
3. If the number of valid connection matrices is less than 1000: try creating both eager and lazy encoders. Otherwise, try creating lazy encoders only. If any encoder is created, select the best encoder, considering only the first 4 priority areas. If successful, use the selected encoder and quit the process.
4. If only lazy encoders were tried at the last step, try eager encoders and select the best, considering all priority areas. If successful, use the selected encoder and quit the process.
5. Try creating ordinal encoders, and select the best encoder from all created encoders, considering all priority areas.

Selection results are cached, because the complete selection process can take a long time, especially if the DSG contains multiple connection choices.

D
DSG, ADSG and ADORE Model Layers

ADORE uses the ADSG, which is a layer on top of the DSG, to model the architecture design space, formulate optimization problems, and convert design vectors to architecture instances. Figure D.1 shows the detailed interactions between these three layers, and how the layers are involved in the architecture optimization loop. It also shows how the System Architecture Optimization (SAO) loop shown in Figure 3.1 is implemented by ADORE.

The ADORE GUI edits an ADORE model, which is used to construct an ADSG. The ADSG is a DSG, which consists of a set of nodes that are subject to choices and constraints. The DSG is encoded as an optimization problem, which is then represented in ADORE by a design problem. The design problem is instantiated by an evaluator, which also provides the evaluation function. The SBArchOpt problem instance represents the design problem using the SBArchOpt problem API, and is used to run the SBArchOpt optimization algorithm. The optimization loop then consists of converting the design vector generated by the optimizer to an architecture graph (ADSG instance), which is further converted to an architecture instance (ADORE model instance). This then provides input to the evaluation function, which has performance metrics as outputs. The performance metrics are finally provided back to the optimizer, interpreted as objectives and constraints.

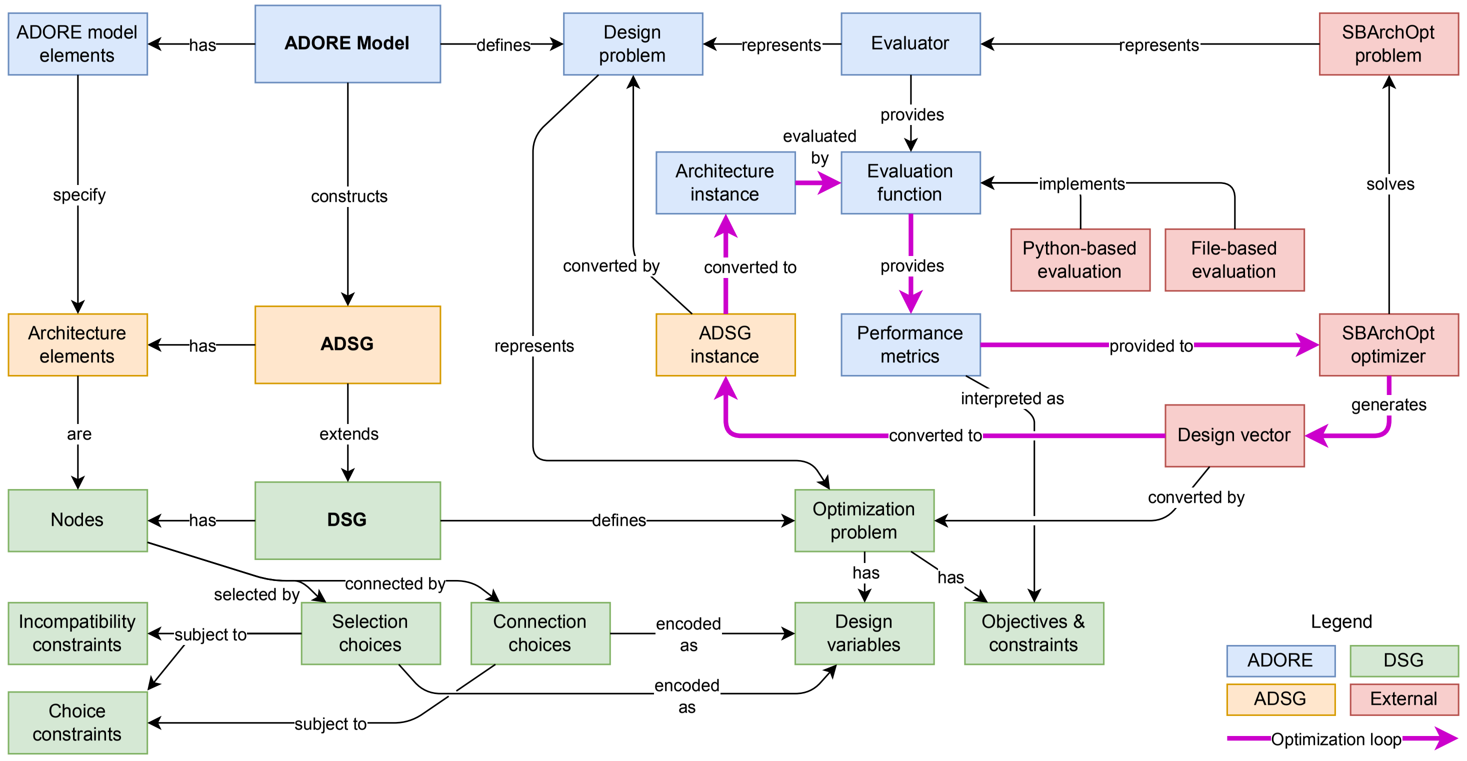


Figure D.1: Connection between DSG, ADSG and ADORE layers and how these are involved in the architecture optimization loop.

E
Optimization Algorithm Performance Comparison

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, calculated at iteration i by:

$$\Delta \mathrm{HV}_i = \frac{\mathrm{HV}_{\mathrm{true}} - \mathrm{HV}_i}{\mathrm{HV}_{\mathrm{true}}}, \tag{E.1}$$

where $\mathrm{HV}_{\mathrm{true}}$ is the hypervolume [315] of the true Pareto front (which is only available for test problems), and HV_i is the hypervolume of the Pareto front at the current algorithm iteration. To compare multiple optimization runs with different initial Design of Experiments (DoEs) the ΔHV ratio is used:

$$\Delta \mathrm{HV}_{\mathrm{ratio},i} = \frac{\Delta \mathrm{HV}_i}{\Delta \mathrm{HV}_0}, \tag{E.2}$$

where $\Delta \mathrm{HV}_0$ is the value at the first iteration (i.e. after the DoE has been evaluated). Regret represents the cumulative error, in our case $\Delta \mathrm{HV}_{\mathrm{ratio}}$, over the course of an optimization [167]: lower values are better, and the value gives an indication of both how closely the optimum is approached by the end of the optimization and by how quickly this was achieved. ΔHV regret at iteration i is calculated from:

$$\mathrm{Regret}(\Delta \mathrm{HV})_i = \mathrm{Regret}(\Delta \mathrm{HV})_{i-1} + \frac{1}{2} n_{\mathrm{step}} \left(\Delta \mathrm{HV}_{\mathrm{ratio},i-1} + \Delta \mathrm{HV}_{\mathrm{ratio},i} \right), \tag{E.3}$$

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

This chapter is based on [4].

Ranking Procedure 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[1] function. The algorithm for determining rank is listed in Algorithm 4. 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.

Algorithm 4 Determine performance rank R_i for each algorithm configuration C_i given performance measure p_i with standard deviation σ_i.

```
Require: $C$, $p$, $\sigma$, $n_{\text{samples}}$, perfMin
Ensure: $R$
 1: $R \leftarrow zeros$                                   ▷ Initialize ranks to 0 (unevaluated)
 2: if perfMin then                                     ▷ Get initial best performing configuration
 3:     $i_{\text{comp}} \leftarrow \arg\min p$
 4: else
 5:     $i_{\text{comp}} \leftarrow \arg\max p$
 6: end if
 7: $R_{i_{\text{comp}}} \leftarrow 1$                      ▷ Set initial best performing configuration to rank 1
 8: while any($R = 0$) do                                ▷ Loop while there are unevaluated ranks
 9:     $i_{\text{uneval}} \leftarrow \arg R = 0$                                 ▷ Get unevaluated ranks
10:     if perfMin then                           ▷ Get best performing unevaluated configuration
11:         $j_{\text{comp}} \leftarrow \arg\min p(i_{\text{uneval}})$
12:     else
13:         $j_{\text{comp}} \leftarrow \arg\max p(i_{\text{uneval}})$
14:     end if
15:     $p_{\text{same}} \leftarrow \text{tTestInd}(p_{i_{\text{comp}}}, \sigma_{i_{\text{comp}}}, n_{\text{samples}}, p_{j_{\text{comp}}}, \sigma_{j_{\text{comp}}}, n_{\text{samples}})$
16:     if $p_{\text{same}} \le 10\%$ then                       ▷ If probability of being the same is low
17:         $R_{j_{\text{comp}}} \leftarrow R_{i_{\text{comp}}} + 1$            ▷ Assign higher rank to compared configuration
18:         $i_{\text{comp}} \leftarrow j_{\text{comp}}$                                ▷ Update reference configuration
19:     else
20:         $R_{j_{\text{comp}}} \leftarrow R_{i_{\text{comp}}}$                                    ▷ Assign same rank
21:     end if
22: end while
```

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

E.1. INTERPRETATION OF RESULTS TABLES

Table E.1 shows an example table presenting the results of a performance comparison and ranking. Such a table can be interpreted as follows:

- The rows list the configurations that are being compared to each other.
- The columns (after the first, and before the "rank" columns) list the test problems the configurations are compared on.
- Per problem (i.e. column), the configuration performances are ranked using the previously described procedure. The best performing configurations are assigned a rank of 1, higher ranks are assigned to worse performing configurations.
- The "rank" columns express for a given configuration how often (seen over all test problems) the configuration achieves these ranks: "rank 1" expresses how often rank 1 is achieved, "rank ≤ 2" expresses how often rank 1 or 2 is achieved.
- The best configurations are selected by first selecting the set of configurations that achieve rank ≤ 2 most often, and then within that set selecting the configurations that achieve rank 1 the most often. The best configurations are underlined.
- The penalty column represents for a given configuration the mean ΔHV regret increase (seen over all test problems) compared to the best configurations. The penalty gives a quantitative estimate of how configurations compare to each other, roughly indicating how much slower to converge and/or how much further away from the Pareto front the configurations are compared to the best configurations.

Table E.1: Example of a table comparing the performance of different optimization configurations on a set of test problems ("A" to "H"), ranked by ΔHV regret (lower rank is better). The best performing configuration is underlined; darker colors represent better results. Penalty represents the mean ΔHV regret increase compared to the best configuration.

	Test Problems										
Config	A	B	C	D	E	F	G	H	Rank 1	Rank ≤ 2	Penalty
Config 1	2	1	2	5	4	3	6	4	12%	38%	12%
Config 2	6	5	6	2	1	8	3	1	25%	38%	455%
Config 3	1	2	1	5	5	1	6	2	38%	62%	0%
Config 4	2	2	3	3	3	3	4	2	0%	38%	9%
Config 5	4	4	5	1	2	6	1	1	38%	50%	81%
Config 6	1	2	2	6	6	2	6	3	12%	50%	7%
Config 7	2	1	3	4	4	4	4	1	25%	38%	15%
Config 8	4	4	5	1	2	6	1	1	38%	50%	84%
Config 9	1	2	1	4	4	3	5	1	38%	50%	6%
Config 10	3	3	4	3	3	5	3	1	12%	12%	25%
Config 11	5	4	5	1	2	7	2	1	25%	50%	118%

F
Multi-stage Launch Vehicle CDS and NFE

Example input file showing the CDS used in the multi-stage launch vehicle MDAO problem:

```
<Rocket>
    <Stage UID="stage_1">
        <Engines>
            <Solid>
                <SRB/>
            </Solid>
        </Engines>
        <Geometry>
            <Length>32.76</Length>
        </Geometry>
    </Stage>
    <Stage UID="stage_2">
        <Engines>
            <Liquid>
                <SIVB/>
            </Liquid>
        </Engines>
        <Geometry>
            <Length>22.39</Length>
        </Geometry>
    </Stage>
    <Stage UID="stage_3">
        <Engines>
            <Liquid>
                <RS68/>
            </Liquid>
        </Engines>
        <Geometry>
            <Length>22.53</Length>
        </Geometry>
    </Stage>
    <Geometry>
        <Head_shape>Elliptical</Head_shape>
        <L_ratio_ellipse>0.175</L_ratio_ellipse>
        <L_D>17.55</L_D>
    </Geometry>
    <Structure>
        <Max_q>50000.0</Max_q>
    </Structure>
    <Payload>
        <Density>2810.0</Density>
    </Payload>
</Rocket>
```

Static input file containing NFE factories at `/Rocket/nfeSettings`:

```
<Rocket>
    <nfeSettings>
        <factories>
            <factory><!-- Rocket stage length -->
                <element>
                    <name>Rocket Body</name>
                    <match>auto</match>
                </element>
                <!-- Write in Stage with a UID containing the stage system index -->
                <xpath>/Rocket/Stage[@UID="stage_{SYS_INDEX}"]</xpath>
                <!-- Create a Geometry node with Length child node -->
                <tag>Geometry</tag>
                <childNodes>
                    <!-- Length child node contains value of the Length design variable -->
                    <Length><element>
                        <name>Length</name>
                        <match>auto</match>
                        <type>QOI</type>
                    </element></Length>
                </childNodes>
            </factory>

            <!-- Stage engines -->
            <factory>
                <element>
                    <name>VULCAIN</name>
                    <match>auto</match>
                </element>
                <!-- {SYS_INDEX_1} refers to the index of the system
                     one system up from the containing system: the "Stage" system -->
                <xpath>/Rocket/Stage[@UID="stage_{SYS_INDEX_1}"]/Engines/Liquid</xpath>
                <tag>VULCAIN</tag>
            </factory>
            <factory>
                <element><name>RS68</name><match>auto</match></element>
                <xpath>/Rocket/Stage[@UID="stage_{SYS_INDEX_1}"]/Engines/Liquid</xpath>
                <tag>RS68</tag>
            </factory>
            <factory>
                <element><name>S_IVB</name><match>auto</match></element>
                <xpath>/Rocket/Stage[@UID="stage_{SYS_INDEX_1}"]/Engines/Liquid</xpath>
                <tag>SIVB</tag>
            </factory>
```

```
<factory>
    <element><name>SRB</name><match>auto</match></element>
    <xpath>/Rocket/Stage[@UID="stage_{SYS_INDEX_1}"]/Engines/Solid</xpath>
    <tag>SRB</tag>
</factory>
<factory>
    <element><name>P80</name><match>auto</match></element>
    <xpath>/Rocket/Stage[@UID="stage_{SYS_INDEX_1}"]/Engines/Solid</xpath>
    <tag>P80</tag>
</factory>
<factory>
    <element><name>GEM60</name><match>auto</match></element>
    <xpath>/Rocket/Stage[@UID="stage_{SYS_INDEX_1}"]/Engines/Solid</xpath>
    <tag>GEM60</tag>
</factory>

<factory><!-- Head geometry: cone -->
    <element><name>Cone</name><match>auto</match></element>
    <xpath>/Rocket</xpath><tag>Geometry</tag>
    <childNodes>
        <Head_shape>Cone</Head_shape>
        <Cone_angle><element>
            <name>Cone Angle</name><match>auto</match><type>QOI</type>
        </element></Cone_angle>
    </childNodes>
</factory>
<factory><!-- Head geometry: semi-sphere -->
    <element><name>Semi-sphere</name><match>auto</match></element>
    <xpath>/Rocket</xpath><tag>Geometry</tag>
    <childNodes>
        <Head_shape>Sphere</Head_shape>
    </childNodes>
</factory>
<factory><!-- Head geometry: elliptical -->
    <element><name>Elliptical</name><match>auto</match></element>
    <xpath>/Rocket</xpath><tag>Geometry</tag>
    <childNodes>
        <Head_shape>Elliptical</Head_shape>
        <L_ratio_ellipse><element>
            <name>Length ratio</name><match>auto</match><type>QOI</type>
        </element></L_ratio_ellipse>
    </childNodes>
</factory>

<factory><!-- Rocket length-to-diameter ratio -->
    <element><name>L to D ratio</name><match>auto</match>
        <type>QOI</type></element>
    <xpath>/Rocket/Geometry</xpath>
    <tag>L_D</tag>
    <!-- Set the linked QOI value as node value -->
    <value><special>VALUE</special></value>
</factory>

<!-- Static inputs -->
<factory>
    <element><name>Max q</name><match>auto</match>
        <type>QOI</type></element>
```

```
                <xpath>/Rocket/Structure</xpath><tag>Max_q</tag>
                <value><special>VALUE</special></value>
            </factory>
            <factory>
                <element><name>Payload density</name><match>auto</match>
                    <type>QOI</type></element>
                <xpath>/Rocket/Payload</xpath><tag>Density</tag>
                <value><special>VALUE</special></value>
            </factory>

        </factories>

        <metrics><!-- Link the output metrics -->
            <metric><!-- Payload mass -->
                <qoi>
                    <name>Payload Mass</name>
                    <match>auto</match>
                </qoi>
                <xpath>/Rocket/Payload/Mass</xpath>
            </metric>
            <metric><!-- Cost -->
                <qoi><name>Cost</name><match>auto</match></qoi>
                <xpath>/Rocket/Cost/Total_cost</xpath>
            </metric>
            <metric><!-- Structural constraint -->
                <qoi><name>Structural Constraint</name><match>auto</match></qoi>
                <xpath>/Rocket/Structure/Constraint</xpath>
            </metric>
            <metric><!-- Volume constraint -->
                <qoi><name>Volume Constraint</name><match>auto</match></qoi>
                <xpath>/Rocket/Payload/Constraint</xpath>
            </metric>
        </metrics>
    </nfeSettings>
</Rocket>
```

CURRICULUM VITÆ

Jasper Herm Bussemaker was born on 29 July 1993 in Amsterdam, the Netherlands. In 2005, his father introduced him to Visual Basic 6 and taught him how to program. During high school, Jasper discovered the world of aviation and spent many hours simulating flights with Microsoft Flight Simulator. In 2011, he obtained his A-Levels (VWO) high school diploma at the Spinoza Lyceum in Amsterdam.

The choice of what to study was not hard, as Aerospace Engineering elegantly combined his passions for programming and aviation. He completed his BSc in 2015 and his MSc in Flight Performance & Propulsion in 2018, both at the Faculty of Aerospace Engineering of Delft University of Technology (TU Delft). His studies included a semester abroad at the University of Kansas (KU), the 1$^{\text{st}}$ place with his group at the Design Synthesis Exercise (the BSc capstone project) symposium, an internship at the Netherlands Aerospace Center (NLR) in Amsterdam as part of the Honours Programme Masters, and an internship at Airbus in Hamburg, Germany. He wrote his master's thesis at Airbus in Munich, Germany about integrating active load control in multidisciplinary optimization codes for high aspect ratio transport aircraft wings. Next to studying, he ran his own company creating websites as a freelancer, and worked as a full-stack developer at EXO-L.

After his studies, Jasper returned to Hamburg to work as a researcher on the topic of Multidisciplinary Design Analysis and Optimization (MDAO) in the team of Pier Davide Ciampa at the Institute of System Architectures in Aeronautics of the German Aerospace Center (DLR). From 2019 to 2023, he worked on the EU Horizon 2020 funded AGILE 4.0 project, for which the capability to "automatically generate system architectures" was needed. Evolving from there, Jasper became an external PhD candidate at TU Delft Aerospace Engineering on the topic of System Architecture Optimization (SAO) in 2021, under the joint supervision of Gianfranco La Rocca (TU Delft) and Nathalie Bartoli (ONERA / ISAE-SUPAERO). In 2023, he spent three months at ONERA in Toulouse, France to work on optimization algorithms in the DTIS department with Nathalie and others. He is a member of INCOSE through its German chapter GfSE since 2023. His paper about hidden constraint strategies presented at the AIAA AVIATION conference in Las Vegas, NV in 2024 received an award for the "Best Student Paper in Multidisciplinary Design Optimization". His paper about SAO for space missions presented at the MBSE2024 workshop in Bremen, Germany received the "Best Paper Award".

Currently, Jasper continues developing SAO capabilities, investigating its integration in Model-Based Systems Engineering (MBSE) processes, and applying SAO to aerospace and non-aerospace problems in various national and international research projects. Next to spending time with his family and friends, he enjoys flying in the Airbus flight club from the Airbus Finkenwerder factory site, where he obtained his EASA PPL(A) in 2021. Jasper lives in Hamburg with his wife Melissa, son Carlo, and two cats Elmi & Jip.

List of Publications and Software

This dissertation presents the results of six years of research, most of which has been disseminated in various publications during this time. The following sections list all publications of which Jasper was at least a co-author. The "primary publications" list contains the papers forming the direct basis for this dissertation, separated into peer-reviewed and non-peer-reviewed publications. The "software list" contains all software that the author has contributed to in the context of this dissertation. The "other publications" list contains papers that publish early versions of parts of the research work, demonstrate various aspects of the research work, or disseminate other project results. Paper [7] received the "Best Student Paper Award in Multidisciplinary Design Optimization" at the AIAA AVIATION Forum 2024. Paper [17] received the "Best Paper Award" at the MBSE2024 Workshop: Model-Based Space Systems and Software Engineering in Bremen, Germany, where it was originally presented.

Primary Publications: Peer-reviewed

[1] J.H. Bussemaker and P.D. Ciampa. “MBSE in Architecture Design Space Exploration”. In: *Handbook of Model-Based Systems Engineering*. Ed. by A.M. Madni, N. Augustine, and M. Sievers. Switzerland: Springer, Apr. 2022. DOI: 10.1007/978-3-030-27486-3_36-1.

[2] J.H. Bussemaker et al. “Function-Based Architecture Optimization: An Application to Hybrid-Electric Propulsion Systems”. In: *33rd Annual INCOSE International Symposium*. Honolulu, HI, USA, July 2023. DOI: 10.1002/iis2.13020.

[3] J.H. Bussemaker. “SBArchOpt: Surrogate-Based Architecture Optimization”. In: *Journal of Open Source Software* 8.89 (Sept. 2023), p. 5564. DOI: 10.21105/joss.05564.

[4] J.H. Bussemaker et al. “System Architecture Optimization Strategies: Dealing with Expensive Hierarchical Problems”. In: *Journal of Global Optimization* (Nov. 2024). DOI: 10.1007/s10898-024-01443-8.

Primary Publications: Non-peer-reviewed

[5] J.H. Bussemaker, P.D. Ciampa, and B. Nagel. “System Architecture Design Space Exploration: An Approach to Modeling and Optimization”. In: *AIAA AVIATION 2020 FORUM*. Virtual Event, June 2020. DOI: 10.2514/6.2020-3172.

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

[7] J.H. Bussemaker et al. “Surrogate-Based Optimization of System Architectures Subject to Hidden Constraints”. In: *AIAA AVIATION 2024 FORUM*. Las Vegas, NV, USA, July 2024. DOI: 10.2514/6.2024-4401.

[8] J.H. Bussemaker, L. Boggero, and B. Nagel. “System Architecture Design Space Exploration: Integration with Computational Environments and Efficient Optimization”. In: *AIAA AVIATION 2024 FORUM*. Las Vegas, NV, USA, July 2024. DOI: 10.2514/6.2024-4647.

[9] S. Garg et al. “Dynamic Formulation and Excecution of MDAO Workflows for Architecture Optimization”. In: *AIAA AVIATION 2024 FORUM*. Las Vegas, NV, USA, July 2024. DOI: 10.2514/6.2024-4402.

SOFTWARE LIST

Name	Contribution	Availability	Section	References
SMT	Hierarchical design space implementation	Open-source[1]	2.3.4	[26]
SBArchOpt	Main developer	Open-source[2]	2.5	[3, 4, 7]
ADSG Core	Main developer	Open-source[3]	3.1	[8]
ADORE	Main developer	Proprietary	3.3	[5, 8, 12]
MDAx	Co-developer	Proprietary	4.1.3	[9, 19]

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

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

[3] https://adsg-core.readthedocs.io/

OTHER PUBLICATIONS

[10] J.H. Bussemaker et al. “Effectiveness of Surrogate-Based Optimization Algorithms for System Architecture Optimization”. In: *AIAA AVIATION 2021 FORUM*. Virtual Event, Aug. 2021. DOI: 10.2514/6.2021-3095.

[11] J.H. Bussemaker et al. “System Architecture Optimization: An Open Source Multidisciplinary Aircraft Jet Engine Architecting Problem”. In: *AIAA AVIATION 2021 FORUM*. Virtual Event, Aug. 2021. DOI: 10.2514/6.2021-3078.

[12] J.H. Bussemaker, L. Boggero, and P.D. Ciampa. “From System Architecting to System Design and Optimization: A Link Between MBSE and MDAO”. In: *32nd Annual INCOSE International Symposium*. Detroit, MI, USA, June 2022. DOI: 10.1002/iis2.12935.

[13] J.H. Bussemaker et al. “Collaborative Design of a Business Jet Family Using the AGILE 4.0 MBSE Environment”. In: *AIAA AVIATION 2022 FORUM*. Chicago, USA, June 2022. DOI: 10.2514/6.2022-3934.

[14] J.H. Bussemaker and L. Boggero. “Technologies for Enabling System Architecture Optimization”. In: *ONERA-DLR Aerospace Symposium (ODAS) 2022*. June 2022.

[15] J.H. Bussemaker, S. Garg, and L. Boggero. “The Influence of Architectural Design Decisions on the Formulation of MDAO Problems”. In: *3rd European Workshop on MDO*. Paris, France, Sept. 2022.

[16] J.H. Bussemaker, L. Boggero, and B. Nagel. “The AGILE 4.0 Project: MBSE to Support Cyber-Physical Collaborative Aircraft Development”. In: *INCOSE International Symposium* 33.1 (July 2023), pp. 163–182. DOI: 10.1002/iis2.13015.

[17] J.H. Bussemaker and T. Firchau. “System Architecture Optimization: An Example Application to Space Mission Planning”. In: *CEAS Space Journal* (May 2025). Article accepted for publication. DOI: 10.1007/s12567-025-00626-7.

[18] J.H. Bussemaker. *System Architecture Optimization Experiments Dataset*. Feb. 2024. DOI: 10.5281/zenodo.10683733.

[19] A. Page-Risueño et al. “MDAx: Agile Generation of Collaborative MDAO Workflows for Complex Systems”. In: *AIAA AVIATION 2020 FORUM*. Virtual Event, June 2020. DOI: 10.2514/6.2020-3133.

[20] C. Cabaleiro de la Hoz et al. “Environmental and Flight Control System Architecture Optimization from a Family Concept Design Perspective”. In: *AIAA AVIATION 2020 FORUM*. Virtual Event, June 2020. DOI: 10.2514/6.2020-3113.

[21] E.H. Baalbergen et al. “Advancing Cross-Organizational Collaboration in Aircraft Development”. In: *AIAA AVIATION 2022 FORUM*. Chicago, USA, June 2022. DOI: 10.2514/6.2022-4052.

[22] R. Grapin et al. “Regularized Infill Criteria for Multi-objective Bayesian Optimization with Application to Aircraft Design”. In: *AIAA AVIATION 2022 Forum*. American Institute of Aeronautics and Astronautics, June 2022. DOI: 10.2514/6.2022-4053.

[23] A. Jeyaraj et al. “Systems Architecting: A Practical Example of Design Space Modeling and Safety-Based Filtering within the AGILE4.0 Project”. In: *33rd Congress of the International Council of the Aeronautical Sciences, ICAS 2022*. Stockholm, Sweden, Sept. 2022.

[24] M. Fouda et al. “Automated Hybrid Propulsion Model Construction for Conceptual Aircraft Design and Optimization”. In: *33rd Congress of the International Council of the Aeronautical Sciences, ICAS 2022*. Stockholm, Sweden, Sept. 2022.

[25] L. Boggero et al. “The MBSE competence at the German Aerospace Center”. In: *INCOSE International Symposium* 33.1 (July 2023), pp. 910–924. DOI: 10.1002/iis2.13061.

[26] P. Saves et al. “SMT 2.0: A Surrogate Modeling Toolbox with a focus on hierarchical and mixed variables Gaussian processes”. In: *Advances in Engineering Software* 188 (Dec. 2023), p. 103571. DOI: 10.1016/j.advengsoft.2023.103571.

[27] P. Saves et al. “System-of-systems Modeling and Optimization: An Integrated Framework for Intermodal Mobility”. In: *ONERA-DLR Aerospace Symposium (ODAS) 2024*. Braunschweig, Germany, June 2024.

[28] S. Garg et al. “MDAx: Enhancements in a Collaborative MDAO Workflow Formulation Tool”. In: *34th Congress of the International Council of the Aeronautical Sciences, ICAS 2024*. Florence, Italy, Sept. 2024.

[29] Y. Ghanjaoui et al. “An Ontology-Based Approach for the Co-Development and Optimization of Aircraft Cabin Design and Assembly Architectures”. In: *34th Congress of the International Council of the Aeronautical Sciences, ICAS 2024*. Florence, Italy, Sept. 2024.

[30] L. Boggero et al. “Processes, Methods and Tools Supporting the Development of Aeronautical Systems”. In: *34th Congress of the International Council of the Aeronautical Sciences, ICAS 2024*. Florence, Italy, Sept. 2024.

TU Delft Delft University of Technology

isae Institut Supérieur de l'Aéronautique et de l'Espace SUPAERO

DLR Deutsches Zentrum für Luft- und Raumfahrt German Aerospace Center